



UvA-DARE (Digital Academic Repository)

On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges

Poss, R.C.

Publication date
2012

[Link to publication](#)

Citation for published version (APA):

Poss, R. C. (2012). *On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Part IV

Appendices

Appendix A

Information sources for the hardware interface

The description work in chapters 3 and 4 was bootstrapped by a preliminary analysis of what information was already available. The following is an exhaustive list of the sources related to the hardware design at that point (late 2008):

- ⟨i⟩ a library of 24 short example programs written in assembly source, listed in table A.1;
- ⟨ii⟩ 17 academic publications containing descriptions of the hardware architecture, listed in table A.2;
- ⟨iii⟩ an internal (unpublished) technical report describing the machine instructions that control multithreading [LB08];
- ⟨iv⟩ program source code for “`mtsim`,” a functional emulator of an abstract multi-core system using the proposed architecture, using assembly source as input;
- ⟨v⟩ program source code for “`mtaxpasm`,” a translator from assembly source to machine code suitable for use with `mtsim`,
- ⟨vi⟩ program source code for “MGSim,” a cycle-accurate component-level simulation of a multi-core system using the proposed architecture, using binary machine code as input;
- ⟨vii⟩ program source code for a modified version of the GNU assembler and linker¹, which superseded `mtaxpasm` to target MGSim;
- ⟨viii⟩ the author of MGSim, the author of `mtsim`, and the other research staff in charge of the architecture design.

When faced with conflicting information, we investigated the MGSim machine emulator at that time, advertised as the “reference implementation,” to resolve the inconsistencies. This ensures our description is consistent with [Lan07, Lan1x]. Otherwise, the information in sections 3.2 to 3.4 was obtained from sources ⟨ii⟩, ⟨vi⟩ and ⟨viii⟩. The information in chapter 4 was obtained from sources ⟨i⟩ to ⟨iii⟩ and ⟨vi⟩ to ⟨viii⟩.

¹<http://www.gnu.org/software/binutils/>

Program	Description
fft/ fft_seq_u	unoptimized sequential 1D FFT kernel
fft/ fft_seq_o	“ hand-optimized
fft/ fft_mt_u	“ unoptimized, multithreaded
fft/ fft_mt_o	“ hand-optimized
fibo/ fibo	multithreaded program that computes the Nth element of the Fibonacci sequence
livermore/ l1_hydro	multithreaded hydro fragment from the Livermore suite [McM86] (integers only)
livermore/ l2_iccq	multithreaded incomplete cholesky conjugate gradient (integers only)
livermore/ l3_innerprod	multithreaded vector-vector product (integers only)
livermore/ l4_bandedlineareq	multithreaded banded linear equations kernel (integers only)
livermore/ l5_tridiagelim	multithreaded tri-diagonal elimination below diagonal (integers only)
livermore/ l6_genlinrecreq	multithreaded general linear recurrence equations (integers only)
matmul/ matmul0	square matrix-matrix multiplication, integers only, sequential
matmul/ matmul1	“ multithreaded, one level of threads
matmul/ matmul2	“ multithreaded, two levels of threads
matmul/ matmul3	“ multithreaded, three levels of threads
sine/ sine_seq_u	unoptimized, sequential sine function using 9-level Taylor expansion
sine/ sine_seq_o	“ hand-optimized
sine/ sine_mt_u	“ unoptimized, multithreaded
sine/ sine_mt_o	“ hand-optimized
sac/ sac_flat	sequential test program containing an heterogeneous floating-point array computation, hand-compiled from a higher-level functional language
sac/ sac_nested_local	“ multithreaded, using one core only
sac/ sac_nested_group	“ using multiple cores

Table A.1: Example programs from the architecture test suite, December 2008

Key	Title
[BJM96]	Dynamic scheduling in RISC architectures.
[JL00]	Micro-threading: a new approach to future RISC.
[Jes01]	Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines.
[LJ02]	Performance of a micro-threaded pipeline.
[Jes03]	Multi-threaded microprocessors – evolution or revolution.
[Jes04]	Scalable instruction-level parallelism.
[Jes05]	Microgrids – the exploitation of massive on-chip concurrency.
[BJ05]	The challenges of massive on-chip concurrency.
[BHJ06b]	Instruction level parallelism through micro-threading – scalable approach to chip multiprocessors.
[BBG ⁺ 06]	A microthreaded architecture and its compiler.
[Jes06b]	Microthreading, a model for distributed instruction-level concurrency.
[BJK07]	Strategies for compiling μ TC to novel chip multiprocessors.
[ZJ07]	On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores.
[Lan07]	Developing a Reference Implementation for a Microgrid of Microthreaded Microprocessors.
[Jes08a]	Operating systems in silicon and the dynamic management of resources in many-core chips.
[Jes08b]	A model for the design and programming of multi-cores.
[BBG ⁺ 08]	A general model of concurrency and its implementation as many-core dynamic RISC processors.

Table A.2: Academic publications explaining the architecture, December 2008

Appendix B

Optimal control word size analysis

Number of instructions per control word	Size of control word	Minimum I-cache line size (non-aligned)	Minimum I-cache line size (aligned)	Unused bytes in line
1	2 bits	5 bytes	8 bytes	3
2	4 bits	9 bytes	16 bytes	7
3	6 bits	13 bytes	16 bytes	3
4	8 bits	17 bytes	32 bytes	15
5	10 bits	22 bytes	32 bytes	10
6	12 bits	26 bytes	32 bytes	6
7	14 bits	30 bytes	32 bytes	2
8	16 bits	34 bytes	64 bytes	30
9	18 bits	39 bytes	64 bytes	25
10	20 bits	43 bytes	64 bytes	21
11	22 bits	47 bytes	64 bytes	17
12	24 bits	51 bytes	64 bytes	13
13	26 bits	56 bytes	64 bytes	8
14	28 bits	60 bytes	64 bytes	4
15	30 bits	64 bytes	64 bytes	0

Table B.1: Impact of control bits on I-cache line utilization.

Assuming 32-bit instruction formats.

Instruction width	Number of instructions per control word	Minimum I-cache line size (aligned)
8 bits	1	2 bytes
8 bits	3	4 bytes
8 bits	6	8 bytes
8 bits	25	32 bytes
8 bits	51	64 bytes
8 bits	102	128 bytes
16 bits	7	16 bytes
16 bits	14	32 bytes
16 bits	455	1024 bytes
24 bits	1	4 bytes
24 bits	315	1024 bytes
32 bits	15	64 bytes
48 bits	5	32 bytes
64 bits	31	256 bytes
64 bits	62	512 bytes

Table B.2: Number of instructions per control word that maximize I-line utilization.

Parameters suitable for minimal I-cache line sizes lower than 1KiB.

Appendix C

Running example with machine code

This appendix illustrates one of the possible machine interfaces described in chapter 4 by looking at a working example.

The first part of the example is given in table C.1. We consider the *structure* of the memory image first. The first bits are control bits (sections 3.2.1 and 4.4 and Appendix B). These are defined 32 at a time, in groups of 2 bits per following instruction. In this example, the hexadecimal value 0x00030400 can be decomposed from LSB to MSB as 0, 0, 0, 0, 0, 1, 0, 0, 3 followed by 7 groups with value 0. The first value is ignored as it refers to the control word itself; the next 8 values (0, 0, 0, 0, 1, 0, 0, 3) are the control bits for the 8 instructions that follow, with values 0 for all instructions except “setlimit” and “mov” which have values 1 and 3, respectively. The value 1 indicates to the pipeline’s fetch unit to switch to a different thread, whereas the value 2 indicates to terminate the thread execution; both values can be combined using a binary OR. Since the “mov” instruction terminates the thread, there are no more instructions afterwards in this thread program. The memory image thus contains

Offset	Machine code (hex)	Textual representation
0x00	00 04 03 00	(control 0x00030400)
0x04	00 00 60 08	allocate \$3
0x08	00 14 e0 47	mov 0, \$0
0x0c	01 34 e0 47	mov 1, \$1
0x10	22 51 40 40	subl \$2, 2, \$2
0x14	5f 00 62 04	setlimit \$3, \$2
0x18	00 00 60 18	setregs \$3, 0, 0, 0, 0
0x1c	0b 00 60 10	cred \$3, 0x48
0x20	1f 04 e3 47	mov \$3, \$31
0x24	1f 04 ff 47	(padding)
0x28	00 00 fe 2f	(padding)
0x2c	1f 04 ff 47	(padding)
0x30	00 00 fe 2f	(padding)
0x34	1f 04 ff 47	(padding)
0x38	00 00 fe 2f	(padding)
0x3c	1f 04 ff 47	(padding)

Table C.1: Memory image for the program “`fib0`”, late 2008

padding at this point to complete the length of the cache line: the fetch unit loads entire cache lines at a time (64 bytes here) in the instruction cache.

We then consider the *semantics* of the memory image. The system initially starts a single thread. The initial program counter is configurable externally to the system (either as a hardware parameter or a pointer in a memory image), and without further specification it default to 0. This seems inconsistent with the program image, because the data at address 0 is the control word. In fact, the fetch unit always automatically skips the first 4 bytes of a cache line when fetching instructions. This implies that the first instruction is really at address 4, in this case “allocate.” Here we may wonder why no synchronizer configuration is present here before the start of the program, as required by section 4.3.3. In fact, there is a special exception for the initial thread: the hardware does not require this synchronizer specification for the first thread and always allocates 31 local (private) synchronizers.

As per section 4.3.1, the “allocate” instruction is the first step of a family creation, and reserves a yet unused *bulk creation context* from a dedicated memory component on chip. It then stores the address of this context into the target operand of the instruction—here the synchronizer with the name “\$3.”

The next two “mov” instructions load the constants 0 and 1 in synchronizers \$0 and \$1, respectively.

The next “subl” instruction subtracts 2 from the value of synchronizer \$2, and stores the result into \$2. This seems nonsensical, since synchronizer \$2 was not set until this point. This is actually a feature of the software emulation platform MGSim: individual processor synchronizers can be set to predefined values before the system is activated. This is a crude way to provide external input to the program¹. All synchronizers that are not predefined this way are initially in the “empty” state. We can confirm this by running the program without input: execution stops at the “subl” instruction, waiting for input which is never defined. If the synchronizer is predefined, execution flows past the instruction.

The next instruction “setlimit” configures (section 4.3.2.3) the limit parameter of the family context reserved by “allocate,” using the result of “subl.” However, we know that this instruction is also marked with the special control bits that indicate the fetch unit to switch to a different thread. This seems nonsensical, as we have seen that initially only one thread is created. Actually, the fetch unit has a special case for the situation where only one thread is active: in that case, the “switch” annotation is ignored and the next instruction is fetched from the same thread. From this point it may become unclear why the “switch” annotation was specified at all. From the same sources we learn that this was done for consistency: *the switch annotation should placed on all instructions that may suspend a thread*, to ensure that the next slots in the pipeline are always occupied with work if there are other threads active.

The next instruction “setregs” configures the offset, in the creating thread’s visible window of synchronizers, of the channel endpoints with the created family. Here we recognize an interface with “hanging” synchronizers (section 4.3.3.3). Here all offsets are set to 0, which means that the first synchronizer name used to communicate will be “\$0” for each channel. The endpoint synchronizer names are chosen contiguously from the first. This instruction clarifies the role of the two preceding “mov” instructions: by placing values in synchronizers “\$0” and “\$1,” they have prepopulated the input side of the communication with the created family. Interestingly, the instruction only specifies one offset for each of the

¹While this is clearly an unrealistic feature for a hardware implementation, it is a simple and convenient system that automates execution of the same program for multiple input values.

channel categories (integer “global,” integer “shared,” floating-point “global,” floating-point “shared”); meanwhile, we know from section 4.3.3.3 that the architecture connects the outgoing “shared” channels of the last thread to the parent thread. In fact, the 2nd and 4th parameter to “setregs” specify *both* the endpoint of the outgoing value to the first thread, and the incoming value from the last thread. Each of these offset specifications may be eventually ignored by the hardware if no channel of the corresponding type is defined.

The next instruction “cred” creates the family of threads using the family context identified by its 1st operand (here \$3, set to the result of “allocate”), and the program counter identified by its 2nd operand (here 0x48). This is a “fused creation” (section 4.3.1.2): the execution of “cred” further binds the 1st operand to a *future* termination value for the created family. This means that the named synchronizer (here \$3) is set to the “empty” state, with the contract that the hardware will change it to the “full” state when all threads in the family terminate. This implies that any further instruction that uses the same synchronizer as input operand will cause the thread to suspend until the family terminates.

Note that both “crei” and “cred” exist; the former accepts a synchronizer operand and an offset for the program counter (“create Indirect”), while the latter accepts an immediate value (“create Direct”). They have otherwise the same semantics, and are collectively named “create.”

The final instruction “mov” reads from the named synchronizer \$3; as seen above this causes the thread to wait until the created family terminates. As per section 4.3.1.2, with a fused creation style, the context reserved by “allocate” is automatically released upon termination and thus made available for subsequent allocations by the same thread or other threads on the processor.

Notwithstanding the question of the validity of address 0x48 in the “cred” instruction, discussed below, we can summarize the following:

- upon system start-up, execution starts with 1 thread using 31 local synchronizers and a configurable initial program counter;
- the synchronizer configuration is omitted for the program of the initial thread;
- the fetch stage skips the first 4 bytes of each cache line;
- control words are organized in groups of 2 bits, one per instruction; value 2 indicates “end thread”, value 1 indicates “switch;”, and the binary OR of both values is possible;
- switch does not really occur in the fetch unit if there is only 1 thread active;
- initially all synchronizers have the state “empty,” except in MGSim where they can be predefined to input values by the user;
- programmers should annotate all instructions that may suspend a thread with the “switch” control bits;
- “cred” and “crei” cause the actual creation of threads for a family; their output in the creating thread is a future on the family’s termination, and the target synchronizer operand of both instructions is “tied up” to the execution of the family until the family terminates (and thus cannot be reused until then).
- “allocate” reserves a family context from a finite-size hardware data structure, used subsequently by “cred” and “crei” and released for further reuses when the family terminates;
- “setlimit” configures the limit parameter for the named family identified by its first input operand, using the value of the 2nd input operand; other examples show that the instructions “setstart,” “setstep” and “setblock” also exist which configure resp. the start, step and “block” parameters for the family (cf. sections 4.3.2.2 and 4.3.2.3);

Offset	Machine code (hex)	Textual representation
0x40	d0 00 00 00	(control 0x000000d0)
0x44	40 00 00 00	(reg. spec. 0x00000040)
0x48	01 00 43 40	addl \$2, \$3, \$1
0x4c	00 04 e3 47	mov \$3, \$0
0x50	1f 04 ff 47	(padding)
0x54	00 00 fe 2f	(padding)
0x58	1f 04 ff 47	(padding)
0x5c	00 00 fe 2f	(padding)
0x60	1f 04 ff 47	(padding)
0x64	00 00 fe 2f	(padding)
0x68	1f 04 ff 47	(padding)
0x6c	00 00 fe 2f	(padding)
0x70	1f 04 ff 47	(padding)
0x74	00 00 fe 2f	(padding)
0x78	1f 04 ff 47	(padding)
0x7c	00 00 fe 2f	(padding)

Table C.2: Continuation of table C.1

- “setregs” configures the channel endpoints in the creating thread for the named family identified by its input operand, using 4 constant values that denote offsets in the thread’s virtual synchronizer window. The 4 offsets specify the integer “global,” integer “shared,” floating-point “global” and floating-point “shared,” in this order; the 2nd and 4th offsets are used both for the outgoing endpoint to the first thread and the incoming endpoint from the last thread.

We can then proceed with the study of the rest of the memory image of the example, listed in table C.2. As expected, there is data in memory at the address 0x48, specified by the “cred” instruction at address 0x18. As previously, we can decompose the structure of this cache line first. The control bits indicate values 0, 0, 1, 3, followed by 12 2-bit blocks with value 0. The first value corresponds to the control word itself, which leaves values 0, 1, 3 for the next instructions in the cache line. Value 1 (switch) applies to the “addl” instruction, whereas 3 (end thread) applies to “mov.” This indicates that “mov” is the last instruction; the rest of the cache line is filled with padding.

As to the semantics, the “cred” instruction indicates that the start of the thread program lies at address 0x48. The synchronizer window configuration immediately precedes, here at address 0x44. This value should be decomposed first in blocks of 16 bits from LSB to MSB, here 0x0040 and 0x0000. The 1st block corresponds to integer synchronizers, the 2nd to floating-point synchronizer. Each block is then further decomposed from LSB to MSB in groups of three 5-bit values, here thus 0, 2, 0 for integers and 0, 0, 0 for floats. In each block, the first value indicates the number of global channels, the 2nd value indicates the number of shared channels, and the 3rd value indicates the number of local (private) synchronizers. In this example the configuration defines only 2 shared channels, i.e. 4 synchronizer endpoints as per section 4.3.3.3 (two incoming, two outgoing).

The first instruction in the thread program is then “addl,” with inputs \$2 and \$3 and output \$1. As this interface maps windows in the order G-S-L-D (section 4.3.3.4), the outgoing channel endpoints are mapped before the incoming endpoints. Since there are no

other synchronizers defined in this example, the names \$0 and \$1 thus point to the outgoing channels, and \$2 and \$3 point to the incoming channels. Therefore, the “addl” instruction can be understood to read from each incoming channel, and produce its output on the 2nd outgoing channel.

The subsequent “mov” instruction then propagates the value from the 2nd incoming channel to the 1st outgoing channel, and the thread then terminates.

Finally, we need to look again at the previous fragment from table C.1 to understand where the endpoints of the first and last links are connected (the outgoing part of the incoming channels of the first thread in the family, and the incoming part of the outgoing channels of the last thread). As described above, they are specified by “allocate,” in this case all endpoints start from the first local synchronizer, here \$0. Therefore, the two endpoints of the first shared channel are mapped onto synchronizer \$0 of the creating thread, and the 2nd shared channel endpoints are mapped onto \$1. It is relevant to observe at this point that these two synchronizers are initialized to 0 and 1 by the creating thread.

From a slightly higher level perspective, we can describe the expected behavior of this program as follows: the program takes an input value, let us call it N . It then creates a family of $N - 2$ threads linked using two daisy-chains of synchronous channels; and initializes the two inputs of the first thread to 0 and 1, respectively. Finally, it waits on termination of the family. Meanwhile, each created thread sums its two inputs into the 2nd output channel, and propagates the 2nd input to the 1st output channel. Inductively, the values produced by the last threads are the N th and $N - 1$ th values of the Fibonacci sequence. It is then possible to observe externally the final state of synchronizers \$0 and \$1 in the creating thread after it terminates to obtain the program’s result.

We can summarize further:

- the first instruction of a thread program cannot be placed directly after the control word, so as to leave room for the synchronizer configuration information which must immediately precede;
- the synchronizer configuration word, placed immediately before the first thread program instruction, is specified as two 16-bit blocks containing 3 5-bit values each, defining the number of global, shared, local synchronizers for integers and floats, in this order;

We can see an example of synchronizer sharing in fig. C.1. This state was obtained after running a thread program with synchronizer configuration 4, 6, 9, 0, 0, 0, which in turn created a family of 3 threads whose synchronizer configuration was 2, 3, 4, 0, 0, 0. The “setregs” operation indicated offsets 1, 5, 0, 0 for the channel endpoints in the creating thread. The shaded areas indicate the resulting mapping of physical synchronizers to logical names in the visible synchronizer windows of each thread; there are 6 unused synchronizer names in the creating thread, and 19 unused synchronizer names in each of the 3 threads in the created family.

In contrast, if the implementation was using separate synchronizers (section 4.3.3.3), as opposed to “hanging” in the previous case, the same configuration would yield the pattern illustrated in fig. C.2.

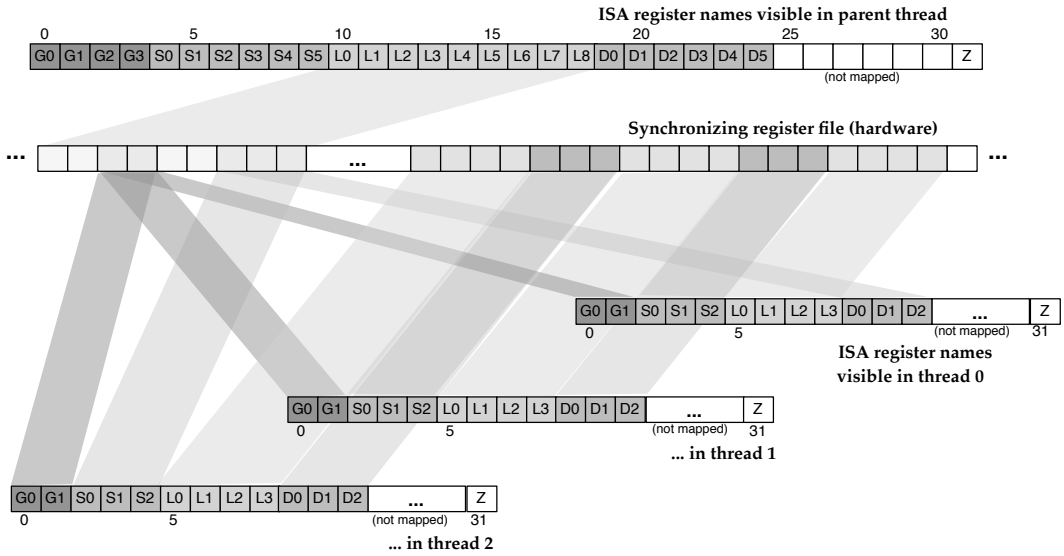


Figure C.1: Observed synchronizer sharing between a creating thread and a family of 3 threads.

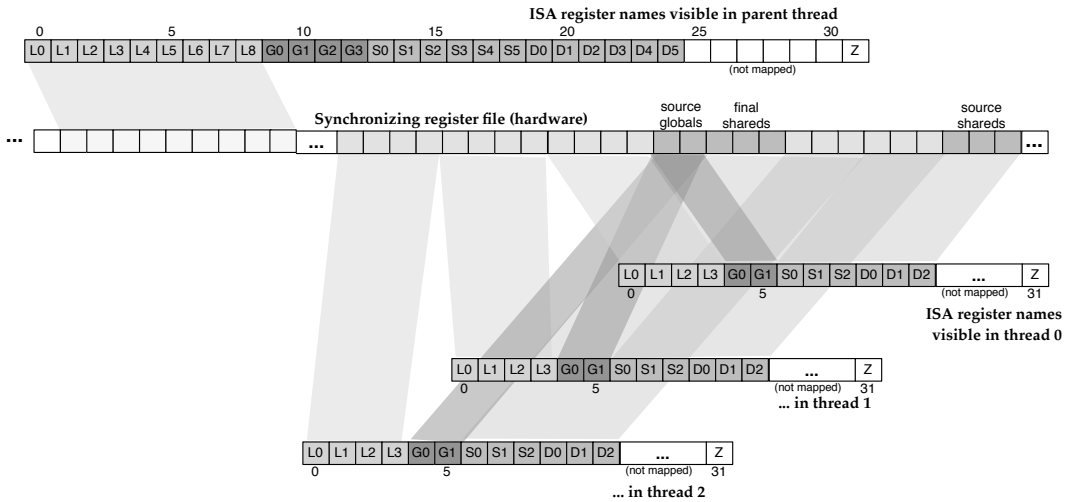


Figure C.2: Observed register sharing between a creating thread and a family of 3 threads.

Appendix D

Assembly instruction formats and machine encoding

Mnemonic	Description	Sync. / Async.	Asynchronous completion condition
crei , cred	Start bulk creation of logical threads	Async.	Bulk creation has started.
sync	Request bulk synchronization.	Async.	All bulk created threads have terminated.
create	Start bulk creation and request bulk synchronization.	Async.	All bulk created threads have terminated [◊]
creba , creba/s , crebas	Issue a combined allocation, configuration and creation request with parameters stored in memory at the specified location.	Async.	Bulk creation has started [†]
crebi , crebi/s , crebis	Issue a combined allocation, configuration and creation request with parameters stored in memory, looked up by index in a table in memory starting at an address stored in a predefined ancillary system register on the local core.	Async.	Bulk creation has started [†]
break	Stop creating new logical threads in the current bulk creation, but let existing logical threads to completion.	Sync.	N/A

[◊] The “**create**” instruction on UTLEON3 implements fused creation as described in section 4.3.1.2.

[†] Only the “*s” variants acknowledge bulk creation. “**creba**” and “**crebi**” do not output a future.

Table D.1: Description of the family control instructions.

Mnemonic	Description	Sync. / Async.	Asynchronous completion condition
<code>putg</code> , <code>fputg</code> , <code>puts</code> , <code>fputs</code>	Write a remote register/synchronizer.	Async.	None: fully asynchronous.
<code>getg</code> , <code>fgetg</code> , <code>gets</code> , <code>fgets</code>	Read a remote register/synchronizer.	Async.	Remote value received [‡]

[‡] The “`get`” instructions to not suspend if the remote synchronizer is not *full* (no dataflow synchronization across cores); instead an undefined value is returned.

Table D.2: Description of the register-to-register communication instructions.

Mnemonic	Description	Sync. / Async.	Asynchronous completion condition
<code>allocate</code> , <code>allocates</code> , <code>allocate/s</code> , <code>allocatex</code> , <code>allocate/x</code>	Allocate a bulk creation context.	Async.	Remote context has been allocated.*
<code>release</code>	Release a previously allocated context	Async.	None: fully asynchronous.
<code>setstart</code> , <code>setlimit</code> , <code>setstep</code> , <code>setblock</code>	Configure the bulk creation parameters after allocation and prior to creation.	Async.	None: fully asynchronous.

* The “`*s`” and “`*x`” variants wait until a context becomes available. The base variant returns a special value to signal if no context was available.

Table D.3: Description of the bulk context management instructions.

Mnemonic	Description
<code>ldbp</code>	Load base pointer: output the base TLS address (cf. chapter 9).
<code>ldfp</code>	Load end pointer: output one address past the end of the TLS space.
<code>gettid</code>	Output the local address of the thread context (within its core).
<code>getfid</code>	Output the local address of the bulk synchronizer (within its core).
<code>getpid</code>	Output the address of the current placement (cf. chapter 11).
<code>getcid</code>	Output the address of the local core.
<code>getasr</code>	Read the content of an ancillary system register on the local core.

Table D.4: Description of miscellaneous instructions.

These instructions retrieve local state relative to the logical thread that issues them.

Assembly format		Encoding							
Pattern	Overlaps/Extends	op1	rd	op3	rs1	i	op _{μT}	ASL _{μT} imm9	rs2
<code>allocate %rd</code>	<code>rdasr %asr20, %rd</code>	10	rd	101000	0x14	0	0x1	-	00000
<code>create %rs2, %rd</code>	<code>rdasr %asr20, %rd</code>	10	rd	101000	0x14	0	0x2	-	rs2
<code>gettid %rd</code>	<code>rdasr %asr20, %rd</code>	10	rd	101000	0x14	0	0x3	-	00000
<code>getfid %rd</code>	<code>rdasr %asr20, %rd</code>	10	rd	101000	0x14	0	0x4	-	00000
<code>launch %rs2</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	0	0x1	-	00000
<code>setstart %rs1, %rs2</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	0	0x2	-	rs2
<code>setstart %rs1, imm9</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	1	0x2	imm9	
<code>setlimit %rs1, %rs2</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	0	0x3	-	rs2
<code>setlimit %rs1, imm9</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	1	0x3	imm9	
<code>setstep %rs1, %rs2</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	0	0x4	-	rs2
<code>setstep %rs1, imm9</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	1	0x4	imm9	
<code>setblock %rs1, %rs2</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	0	0x5	-	rs2
<code>setblock %rs1, imm9</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	1	0x5	imm9	
<code>setthread %rs1, %rs2</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	0	0x6	-	rs2
<code>setthread %rs1, imm9</code>	<code>wrasr %rs1, %asr20</code>	10	0x14	110000	rs1	1	0x6	imm9	
<code>break</code>	<code>wrasr %g0, %asr20</code>	10	0x14	110000	00000	0	0xA	-	00000

Table D.5: MT instruction set extensions implemented in UTLEON3.

Non-prefixed values are in base 2, and values prefixed with “0x” in base 16.

Assembly format		Encoding							
Pattern	Overlaps/Extends	op1	rd	op3	rs1	i	op _{μT}	ASIM _T	rs2
								imm9	
allocate %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x1	-	00000
allocate %rs2, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x1	-	rs2
allocate imm9, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	1	0x1	imm9	
(reserved UTLEON3)	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x2	-	rs2
gettid %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x3	-	00000
getfid %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x4	-	00000
getpid %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x5	-	00000
getcid %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x6	-	00000
crei %rs2, %rs1	rdasr %asr20, %rs1	10	rs1	101000	0x14	0	0x7	-	rs2
cred imm9, %rs1	rdasr %asr20, %rs1	10	rs1	101000	0x14	1	0x7	imm9	
sync %rs2, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x8	-	rs2
allocates %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x9	-	00000
allocates %rs2, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0x9	-	rs2
allocates imm9, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	1	0x9	imm9	
allocatex %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0xA	-	00000
allocatex %rs2, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0xA	-	rs2
allocatex imm9, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	1	0xA	imm9	
gets %rs2, N, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0xB	N	rs2
getg %rs2, N, %rd	rdasr %asr20, %rd	10	rd	101000	0x14	0	0xC	N	rs2
fgets %rs2, N, %fd	rdasr %asr20, %fd	10	fd	101000	0x14	0	0xD	N	rs2
fgetg %rs2, N, %fd	rdasr %asr20, %fd	10	fd	101000	0x14	0	0xE	N	rs2
ldbp %rd	rdasr %asr19, %rd	10	rd	101000	0x13	0	0x1	-	00000
ldfp %rd	rdasr %asr19, %rd	10	rd	101000	0x13	0	0x2	-	00000
crebas %rs2, %rs1	rdasr %asr19, %rs1	10	rs1	101000	0x13	0	0x3	-	rs2
crebas imm9, %rs1	rdasr %asr19, %rs1	10	rs1	101000	0x13	1	0x3	imm9	
crebis %rs2, %rs1	rdasr %asr19, %rs1	10	rs1	101000	0x13	0	0x4	-	rs2
crebis imm9, %rs1	rdasr %asr19, %rs1	10	rs1	101000	0x13	1	0x4	imm9	
(reserved UTLEON3)	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x1	-	00000
setstart %rs1, %rs2	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x2	-	rs2
setstart %rs1, imm9	wrasr %rs1, %asr20	10	0x14	110000	rs1	1	0x2	imm9	
setlimit %rs1, %rs2	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x3	-	rs2
setlimit %rs1, imm9	wrasr %rs1, %asr20	10	0x14	110000	rs1	1	0x3	imm9	
setstep %rs1, %rs2	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x4	-	rs2
setstep %rs1, imm9	wrasr %rs1, %asr20	10	0x14	110000	rs1	1	0x4	imm9	
setblock %rs1, %rs2	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x5	-	rs2
setblock %rs1, imm9	wrasr %rs1, %asr20	10	0x14	110000	rs1	1	0x5	imm9	
(reserved UTLEON3)	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x6	-	rs2
(reserved UTLEON3)	wrasr %rs1, %asr20	10	0x14	110000	rs1	1	0x6	imm9	
release %rs1	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0x9	-	00000
break	wrasr %g0, %asr20	10	0x14	110000	00000	0	0xA	-	00000
puts %rs2, %rs1, N	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0xB	N	rs2
putg %rs2, %rs1, N	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0xC	N	rs2
fputg %fs2, %rs1, N	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0xD	N	fs2
fputs %fs2, %rs1, N	wrasr %rs1, %asr20	10	0x14	110000	rs1	0	0xE	N	fs2
creba %rs1, %rs2	wrasr %rs1, %asr19	10	0x13	110000	rs1	0	0x3	-	rs2
creba %rs1, imm9	wrasr %rs1, %asr19	10	0x13	110000	rs1	0	0x3	imm9	
crebi %rs1, %rs2	wrasr %rs1, %asr19	10	0x13	110000	rs1	0	0x4	-	rs2
crebi %rs1, imm9	wrasr %rs1, %asr19	10	0x13	110000	rs1	0	0x4	imm9	
print %rs1, %rs2	wrasr %rs1, %asr19	10	0x13	110000	rs1	0	0xF	-	rs2
print %rs1, imm9	wrasr %rs1, %asr19	10	0x13	110000	rs1	1	0xF	imm9	

Table D.6: General MT extensions for a SPARC v8 instruction set.

Non-prefixed values are in base 2, and values prefixed with “0x” in base 16.

Assembly pattern	bits 26-31	Format	bits 21-25	bits 16-20	bits 13-15	bit 12	bits 5-11	bits 0-4
ldbp Rc	0x01	Op	11111	11111	0	0	0000000	Rc
ldfp Rc	0x01	Op	11111	11111	0	0	0000001	Rc
gettid Rc	0x01	Op	11111	11111	0	0	0000010	Rc
getfid Rc	0x01	Op	11111	11111	0	0	0000011	Rc
getpid Rc	0x01	Op	11111	11111	0	0	0000100	Rc
getcid Rc	0x01	Op	11111	11111	0	0	0000101	Rc
getasr imm8, Rc	0x01	Op	11111	imm8		1	0000110	Rc
getapr imm8, Rc	0x01	Op	11111	imm8		1	0000111	Rc
break	0x01	Op	11111	11111	0	0	0001000	11111
<i>(future local)</i>	0x01	Op	x	x	x	x	000xxxx	x
setstart Rf, Rv	0x01	Op	Rf	Rv	0	0	0100000	11111
setstart Rf, imm8	0x01	Op	Rf	imm8		1	0100000	11111
setlimit Rf, Rv	0x01	Op	Rf	Rv	0	0	0100001	11111
setlimit Rf, imm8	0x01	Op	Rf	imm8		1	0100001	11111
setstep Rf, Rv	0x01	Op	Rf	Rv	0	0	0100010	11111
setstep Rf, imm8	0x01	Op	Rf	imm8		1	0100010	11111
setblock Rf, Rv	0x01	Op	Rf	Rv	0	0	0100011	11111
setblock Rf, imm8	0x01	Op	Rf	imm8		1	0100011	11111
putg Rv, Rf, imm5	0x01	Op	Rf	Rv	0	0	0100100	imm5
putg imm8, Rf, imm5	0x01	Op	Rf	imm8		1	0100100	imm5
fputg Fv, Rf, imm5	0x05	FP Op	Rf	Fv	0	0	0100101	imm5
puts Rv, Rf, imm5	0x01	Op	Rf	Rv	0	0	0100101	imm5
puts imm8, Rf, imm5	0x01	Op	Rf	imm8		1	0100101	imm5
fputs Fv, Rf, imm5	0x05	FP Op	Rf	Fv	0	0	0100101	imm5
release Rf	0x01	Op	Rf	11111	0	0	0101000	11111
<i>(future remote async)</i>	0x01	Op	x	x	x	x	010xxxx	x
<i>(future remote async)</i>	0x05	FP Op	x	x	x	x	010xxxx	x
sync Rf, Rc	0x01	Op	Rf	11111	0	0	0110000	Rc
getg Rf, imm5, Rc	0x01	Op	Rf	imm5	0	0	0110010	Rc
fgetg Rf, imm5, Fc	0x05	FP Op	Rf	11111	0	0	0110010	Fc
gets Rf, imm5, Rc	0x01	Op	Rf	imm5	0	0	0110011	Rc
fgets Rf, imm5, Fc	0x05	FP Op	Rf	11111	0	0	0110011	Fc
<i>(future remote sync)</i>	0x01	Op	x	x	x	x	011xxxx	Rc
<i>(future remote sync)</i>	0x05	FP Op	x	x	x	x	011xxxx	Fc
allocate Rp, Ro, Rf	0x01	Op	Rp	Ro	0	0	1000000	Rf
allocate Rp, imm8, Rf	0x01	Op	Rp	imm8		1	1000000	Rf
allocate/s Rp, Ro, Rf	0x01	Op	Rp	Ro	0	0	1000001	Rf
allocate/s Rp, imm8, Rf	0x01	Op	Rp	imm8		1	1000001	Rf
allocate/x Rp, Ro, Rf	0x01	Op	Rp	Ro	0	0	1000011	Rf
allocate/x Rp, imm8, Rf	0x01	Op	Rp	imm8		1	1000011	Rf
creba/a Rs, Rv	0x01	Op	Rs	Rv	0	0	1100001	11111
creba/a Rs, imm8	0x01	Op	Rs	imm8		1	1100001	11111
creba/s Rs, Rv, Rf	0x01	Op	Rs	Rv	0	0	1100001	Rf
creba/s Rs, imm8, Rf	0x01	Op	Rs	imm8		1	1100001	Rf
crebi/a Rs, Rv	0x01	Op	Rs	Rv	0	0	1110001	11111
crebi/a Rs, imm8	0x01	Op	Rs	imm8		1	1110001	11111
crebi/s Rs, Rv, Rf	0x01	Op	Rs	Rv	0	0	1110001	Rf
crebi/s Rs, imm8, Rf	0x01	Op	Rs	imm8		1	1110001	Rf
<i>(future allocate)</i>	0x01	Op	x	x	x	x	1xxxxxx	x
crei Rf, imm16(Rp)	0x03	Memory	Rf	Rp	imm16			
cred Rf, imm21	0x04	Branch	Rf	imm21				
print Rs, Rv	0x01	Op	Rs	Rv	0	0	0001111	11111
print Rs, imm8	0x01	Op	Rs	imm8		1	0001111	11111
printf Rs, Fv	0x05	FP Op	Rs	Fv	0	0	0001111	11111

Table D.7: MT extensions for the DEC/Alpha AXP 24264 instruction set.

Non-prefixed values are in base 2, and values prefixed with “0x” in base 16.

Appendix E

On-chip placement and distribution

Abstract

This appendix complements chapter 11 and details the thread mapping of logical threads to processors as implemented in the MGSim system emulator at the end of 2011.

Contents

E.1	General protocol	264
E.2	Reference implementation	265
E.3	SL primitives	266

E.1 General protocol

The requirement for explicit placement is closely related to resource management on chip. Indeed, bulk creation and thread contexts constitute physical resources over one or multiple processors; they are allocated and released via dedicated on-chip messages, which can be sent via processor instructions. Each bulk creation context can be used to activate families composed of concurrent logical threads; logical threads run on a set of parallel thread contexts allocated from physical resources on the same processors as the bulk creation contexts (section 4.2). Both family and thread contexts are thus finite resources local to each processor; a protocol to manually place computation to specific processors is therefore also a protocol that controls resource usage on chip.

We describe the proposed protocol here.

To start with, all execution units (processors, or “cores”) are provided an address on an on-chip network. The “allocate,” “create” and other concurrency operations from chapter 4 should be considered as *event messages* on this network.

A single family allocation event reserves a “virtual” family context spread over multiple cores, which can subsequently be bulk-created and bulk-synchronized. For this the program optionally provides a *placement size* along with the target processor address upon family allocation (section 4.3.2.1). If this size is larger than 1, the processor receiving the allocation may optionally negotiate a multi-core allocation with its neighbors (according to the topology of the network), trying to maximize the number of processors up to the size specified. As with the window size (section 4.3.2.2), parallelism is not guaranteed: the placement size is only an upper bound and the work can be performed even if only one processor is available, or if multi-processor allocation is not supported on that processor, regardless of the size specified.

When a multi-processor allocation completes, a bulk creation context has been reserved on each selected processor and all the contexts are linked to a master context selected among them. This context becomes the result of the original allocation event. When the subsequent creation event is sent to the master context, that processor forwards the creation to all participating processors; conversely, bulk synchronization on the master context effects bulk synchronization on all contexts. The master context is thus a *proxy* and provides the same interface as a single-processor family context to the issuing thread.

To support *implicit placement*, each bulk creation context further holds a default target processor and placement size, which is called its *default placement*. When a bulk creation context is allocated, the default placement is automatically set to the placement specified during allocation. When an allocation is issued by a program without an explicit placement, the default placement is used. This allows a program to automatically restrict the parallelism of a tree of implicitly placed concurrent families to a local cluster around the designated processor, bounded by the placement size.

Finally, the program can also parameterize the *logical thread distribution* of logical threads over processors. At least two distribution strategies are defined. The default is a *flexible distribution* which leaves the distribution unspecified, with the guarantee of a best effort to maximize the amount of parallelism by the specific architecture implementation (e.g. via work stealing). This allows a program to offload load balancing to the underlying hardware implementation. However, when the group of processors is known to be homogeneous, or when an explicit assignment is desired, the program can specify an *even distribution* explicitly which statically partitions the logical threads indexes evenly over the selected processors.

Address used	Automatically transformed to	Message sent to
0	<i>Default placement</i>	Implicitly, same processor cluster as where the issuing thread was placed
1	Processor address of the local processor, size 1, same capability	Implicitly, same processor as where the issuing thread is running
$C + 2P + S$	Processor address P , size S , capability C	Explicit target cluster

Table E.1: Address transformation for allocation messages on chip.

E.1.1 Global family identification

As families can be created across the chip and synchronized on from processors other than where they were created, the protocol defines Global Family Identifiers (GFIDs). These are composed of two parts: the address of the processor where the context is allocated (or the address of the master processor for distributed families) and the address of the context in the local structures of that processor.

To allow flexible control of bulk creation contexts and families by programs across arbitrary points on the chip, the address of a context can be observed and manipulated as a value. Its encoding is constrained to fit in the smallest common machine word size supported throughout the chip, in order to facilitate the integration in a software stack.

E.1.2 Isolation and capabilities

With regards to isolation, the protocol is extensible using capabilities [Lin76]. When these are used, each processor on the system holds a table of *processor access capabilities*, maintained by system-level resource managers, which filters incoming allocation messages. Programs must then request capabilities to said resource managers before they can issue “allocate.” The capabilities need not be visible to programs, as it can be embedded either in a private memory at the requesting core (when dedicating entire cores to an isolated process) or the requesting family or thread context (when fine-grained access control is desired). Upon receiving an allocation message, a core allocation unit then generates a *family access capability* along with the GFID returned to the requesting thread. This must then be joined with the core capability upon subsequent creation, remote register accesses, synchronization and release messages. When a bulk creation context is allocated in one protection domain and used in another, the resource manager must be informed accordingly to propagate the access capabilities.

E.2 Reference implementation

The protocol described above has been prototyped in the system emulator for testing purposes, as follows:

- all the processors are numbered along a space-filling curve to ensure multi-scale space locality on the distributed cache network (fig. 3.9);

Side note E.1: Processor address decoding in the reference implementation.

The addressing scheme requires that all cluster sizes are powers of two, and that a cluster of any size S must start with a processor address multiple of S . The capability bits are placed on the MSBs unused by the processor address.

For example, the value $A = 112040$ on a chip with $N = 64$ processors, encoded in binary as 11011010110101000, corresponds to:

- $S = 8$: find the first bit set from A 's LSB, here 1000,
 - $P = 16$: compute $\frac{A-S}{2} \bmod N$, here 010000,
 - $C = 112000$: compute $A \bmod 2N$, here 11011010110000000.
-

Side note E.2: About the distribution of families with “shared” channels.

For families of threads using “shared” synchronizers (section 4.3.3.3), all threads are always created at the first processor of the target cluster, although their default placement may be configured to the entire cluster. The reason why local threads for this kind of family are not spread over multiple cores stems from an observation when such an implementation was tried. If such families were distributed over multiple processors, the first logical thread which reads from an input “shared” channel on every processor would wait for the corresponding output by from the last logical thread on the previous processor. This would effectively force a sequentialization of the entire communication chain and prevent multi-processor scalability altogether. As this effect was likely to cause programs to avoid using the feature, the extra effort required to implement the feature was not invested.

```

1 static inline size_t sl_placement_size(sl_place_t C)
2 {
3     // erase all bits set but the rightmost, i.e. the size
4     return C & -C;
5 }
```

Listing E.1: Placement computation that extracts the size from a virtual cluster address.

- the placement operand provided to the “allocate” operation (section 4.3.2.1) is handled by the hardware as per table E.1 and side note E.1.
- the implementation supports the even distribution strategy of logical threads to processors in distributed families, but only for families without “shared” channels (cf. side note E.2);
- capabilities are decoded but not verified (yet).

Also, new assembler mnemonics “getpid” and “getcid” are introduced which map to machine instructions that read the default placement and the local processor address of the issuing thread, respectively.

E.3 SL primitives

To control this protocol, the primitives in listings E.1 to E.6 and table E.2 are available in a SL library header file.

Primitive	Alpha instructions	SPARC instructions
<code>sl_placement_size</code>	2	2
<code>sl_first_processor_address</code>	2	2
<code>split_upper</code>	5	5
<code>split_lower</code>	5	5
<code>at_core</code>	7	6
<code>split_sibling</code>	11	11
<code>next_core</code>	13	13
<code>prev_core</code>	13	13

Table E.2: Execution cost of the placement primitives.

All instructions can be performed locally in the ALU in one pipeline cycle.

```

1 static inline sl_place_t sl_first_processor_address(sl_place_t C)
2 {
3     // erase the rightmost bit set, i.e. the size
4     return C & (C - 1);
5 }

```

Listing E.2: Placement computation that extracts the absolute address of the first core in a virtual cluster.

```

1 static inline sl_place_t split_upper(void) {
2     sl_place_t d = sl_default_placement();
3     size_t size = sl_placement_size(d);
4     return d + size / 2;
5 }
6 static inline sl_place_t split_lower(void) {
7     sl_place_t d = sl_default_placement();
8     size_t size = sl_placement_size(d);
9     return d - size / 2;
10 }

```

Listing E.3: Placement computation that divides the current cluster in two and addresses either the upper or lower half.

```

1 static inline sl_place_t at_core(unsigned int P) {
2     sl_place_t d = sl_default_placement();
3     sl_place_t f = sl_first_processor_address(d);
4     return (f + P * 2) | 1;
5 }

```

Listing E.4: Placement computation to place all the created thread at a core offset P within the local cluster.

```

1 static inline sl_place_t split_sibling(void) {
2     sl_place_t d = sl_default_placement();
3     size_t size = sl_placement_size(d);
4
5     sl_place_t lp = sl_local_processor_address();
6     sl_place_t mybit = (lp * 2) & size;
7     sl_place_t otherbit = mybit ^ size;
8
9     sl_place_t f = sl_first_processor_address(d);
10    return f | otherbit | (size / 2);
11 }

```

Listing E.5: Placement computation that divides the current cluster in two and addresses the other half relative to the current core.

```

1 static inline sl_place_t next_core(void) {
2     sl_place_t d = sl_default_placement();
3     size_t size = sl_placement_size(d);
4
5     sl_place_t lp = sl_local_processor_address();
6     // make relative to start of cluster
7     sl_place_t rel_lp = lp & (size - 1);
8
9     // next core address:
10    sl_place_t rel_np = (rel_lp + 1) & (size - 1);
11
12    sl_place_t f = sl_first_processor_address(d);
13    return (f + rel_np * 2) | 1;
14 }
15 static inline sl_place_t prev_core(void) {
16     // same as "nextcore" with
17     // (rel_lp - 1) instead of (rel_lp + 1).
18 }

```

Listing E.6: Placement computation to place all the created thread at the next or previous core within the local cluster.

Appendix F

Semantics of objects in the C language

Abstract

The process of extending the C language for use with a new processor architecture with hardware support for synchronization channels requires us to carefully avoid clashes between the hardware semantics and the (well-specified) semantics of the C abstract machine. As a foundation to this work, we analyzed the specification in [II99, II11b] to uncover and summarize how the C language handles data in programs. To our knowledge, no such analysis exists in previous work. Our findings are detailed here.

Contents

F.1	Fundamental characteristics	270
F.2	Basic operations	271
F.3	Syntax in the C language	272
F.4	Space for optimization	272
F.5	Lexical binding of lifetimes and resource allocation	273
F.6	Primary designators	273
F.7	Secondary designators	273
F.8	The fine print in the C semantics	274

Side note F.1: About the concept of objects in the C language specification.

While objects are a central concepts in the abstract semantics of C, they are only indirectly defined throughout [II99, II11b] and similar documents. An object is any of the following: an entity defined by a declaration, an entity defined by a non-array function parameter declaration, a constant (character/integer, floating-point or literal string), the value of an initializer, the value of an enumeration tag, an allocated object, the result of a computation in an expression, the return value of a function call. The following are not objects despite the possibility of taking their address: functions, labels (in some implementations).

Side note F.2: About immutable objects.

The following objects are immutable: constants, the value of initializers, the value of enumeration tags, temporary results in expressions, return values of function calls, objects defined with the `const` qualifier, non-array function parameters declared with the `const` qualifier; the following objects are mutable: objects defined without the `const` qualifier, non-array function parameters declared without the `const` qualifier.

Side note F.3: About objects without an address.

The following objects cannot have their address taken: objects defined with the `register` storage-class qualifier, temporary results in expressions, return values of function calls, constants, the value of initializers, the value of enumeration tags; the following objects can have their address taken: objects defined without the `register` storage-class qualifier, non-array function parameters, allocated objects.

F.1 Fundamental characteristics

An *object* is an entity able to store data for a C program.

A C object *exists* throughout its *lifetime*.

The following *properties* of objects are *defined and fixed* throughout their lifetime:

- whether the object is *mutable* or not;
- whether the object *can have its address taken* or not;
- its *size*.

The properties of an array considered as an object always extend as properties of its individual items considered as separate objects.

The layout in storage of an object is identical to an array of N objects of type `char` (N being the size of the object as per `sizeof`). This array is the representation of the object.

If an object is mutable, then each `char` of its representation can be muted (modified) individually multiple times, and a read to each will consistently and deterministically yield the value that was last stored before the immediately preceding sequence point. The initial value of each `char` of the representation, although unspecified, exists and can be read.

If the address of an object can be taken, then:

- its address is constant throughout its lifetime,
- it is possible to compute the address of each individual `char` of its representation,
- the addresses of each individual `char` are contiguous,

Side note F.4: About object sizes.

The size of an object is derived from the type used to define the object, or an expression for allocated objects. Note that the size is fixed even for allocated objects: the `realloc` library call semantically returns a new (distinct) object, although its address may be the same as the object given as input.

Side note F.5: About array item properties.

In particular: the items of a mutable array are separately mutable; the items of an immutable array are separately immutable; the items of an array that can have its address taken can have their address taken separately; the items of an array that cannot have its address taken cannot have their address taken separately; the size of an array is the size of an individual item multiplied by the number of items in the array.

Side note F.6: About multiple accesses between sequence points.

If an object is written to, then read between two adjacent sequence points, e.g. in `p[i++] = ++i`, the behavior of the program is undefined.

Side note F.7: About the initial char representation of objects.

The chars of the object representation can be read. In particular, the behavior of the following program piece is well-defined and consistently deterministic: `int x; x = x ^ x;`

- there exist a valid address one past the last char in its representation.

No two distinct mutable objects that can have their address taken have overlapping representations.

It is possible to construct objects of arbitrarily large sizes (as long as matching pointer types, `size_t` and `ptrdiff_t` are provided by the implementation).

F.2 Basic operations

Two basic operations are defined on objects and are valid throughout their lifetime:

- *read* for all objects;
- *write* for mutable objects.

The “read” operation has the following properties:

- it always completes within a finite amount of time before the next sequence point is reached;
- it yields the *value* stored in the object, according to the type of the designator used to access the object.

The “write” operation has the following properties:

- it always completes within a finite amount of time before the next sequence point is reached;
- it modifies the object so that further reads will yield the new value being stored.

Side note F.8: About valid addresses one past the last char.

C provides this guarantee in order to be able to consistently compute sizes by performing pointer arithmetic. A side-effect (arguably harmless) of this guarantee is that no object can occupy the last char in an address space.

It is worthy to note that reading from and writing to an existing object has no influence on the control flow: considering a read and/or write operation, then in all runs of the program where the sequence point immediately before is reached, if the object exists at that point the sequence point immediately after is reached in sequence within a finite amount of time. Of course, if the object does not exist at that point, or if a write operation is attempted on an immutable object, the behavior is undefined.

F.3 Syntax in the C language

The “read” operation that operates on objects is exposed implicitly in the language when objects are designated in expressions.

The “write” operation that operates on objects is exposed by the various assign operators and the increment/decrement operators.

F.4 Space for optimization

The C specification indicates that an expression needs not be evaluated if its value is not used and it does not have side-effects ([II99, 5.1.2.3§3], cite[5.1.2.3§4]isoc11).

This can be reworded as follows:

- if two writes to an object are expressed in a program and it can be proven during compilation that no other read to the same object ever takes place (at run-time) between the two writes, then the program can be transformed to an equivalent program where the first write is omitted; for example:

```

1 x = a; // can be omitted if x is
2       // not read before "x=b"
3       // below.
4 /* ... */
5 x = b;
```

- if two reads to an object are expressed in a program and it can be proven during compilation that no write to the same object ever takes place (at run-time) between the two reads, then the program can be transformed to an equivalent program where the second read is omitted and the value of the first read is reused instead; for example:

```

1 a = x;
2 /* ... */
3 b = x; // can be expressed as
4       // "b=a" if a and x were
5       // not written to.
```

- if a write to an object and a read to the same object are expressed in a program and it can be proven during compilation that no write to the same object ever takes place (at run-time) between the write and the read, then the program can be transformed to an equivalent program where the read is omitted and the value used in the first write is reused; for example:

```

1 a = x;
2 /* ... */
```

Side note F.9: Objects without primary designators.

In some implementations it is possible to define an object with a scoped lifetime, a known type but without a primary designator, e.g. anonymous (unused) function parameter declarations.

Side note F.10: Primary designator aliases for immutable objects.

The C specification leaves space for an implementation to share an immutable object across multiple primary designators, e.g. for string literals.

```

3 b = a; // can be expressed as
4       // "b=x" if a and x
5       // were not written to.
```

This optimization path is used routinely in most production-grade C compilers.

F.5 Lexical binding of lifetimes and resource allocation

The lifetime of an object is either *scoped* or *unscoped*. An object has unscoped lifetime if it is allocated. *All non-allocated objects have scoped lifetimes.*

Scoped lifetimes are bound to a *declaration* (either an object definition or a non-array function parameter declaration) and a *syntactic scope*, and enforced by the environment according to completely specified semantics. Unscoped lifetimes are managed explicitly by the (running) program.

Whether an object with a scoped lifetime exists or not at any given point in a program is *entirely decidable during compilation*. Objects with scoped lifetimes have *all their properties known and fixed by their declaration*. These two properties allow a compiler to *statically allocate resources* for all objects with scoped lifetimes.

F.6 Primary designators

Objects with scoped lifetimes have a type and zero or one *primary designator* (derived from their definition).

No two distinct primary designators that designate mutable objects designate the same object.

As scoping also defines the visibility of primary designators, the portion of a program where a primary designator is visible is always a subset of the lifetime of the object it designates.

The main purpose of the design of primary designators in C is to *uniquely identify* objects.

Objects with unscoped lifetimes do not have a primary designator, always are mutable, always can have their address taken and have a size equal to the value of the argument of the function call used to start their lifetime.

F.7 Secondary designators

Any object can have zero or more *secondary designators* each with their own types (aliases).

For example, in the following program fragment:

Side note F.11: About array designators in function parameter lists.

This is the main distinction between array and non-array function parameter declarations: non-array function parameters are primary designators to distinct objects from the argument in the caller, whereas array function parameters are secondary designators to the same object as the argument in the caller.

```

1 int a;
2 int *p = &a;
3 (a);
4 (*p)

```

There exists one object with primary designator `a` and secondary designator `*p` (and another object with primary designator `p`).

Secondary designators are *constructed* by the program *at run-time*, using only the following operations:

- from an existing designator, using the unary `&` operator;
- from an existing designator, using a cast expression;
- from an existing array designator, using the designator in a context where it is converted automatically to a pointer (either during pointer arithmetic, assignment to pointer, or when passing an array as a function argument);
- from an existing array designator, using the array subscript notation;
- from an existing union or structure designator, using the field subscript notation.

Secondary designators are *used* by the program using only the following syntax:

- for cast notations, using the cast notation directly;
- for field subscript notations, using the field subscript notation directly;
- for array subscript notations, using the array subscript notation directly;
- for secondary designators derived as pointer values (either by the unary `&` or using an array designator in a pointer context), using the unary `*` operator or the array subscript notation on a designator with pointer type to an object containing that pointer value.

The visibility of secondary designators can extend past and before the lifetime of an object. The behavior of a program that references an object via a secondary designator outside of its lifetime is undefined.

Secondary designators can designate any contiguous sequence of chars in the representation of an object; if the object is mutable any modification through a secondary designator only modifies the bytes designated by the designator as constrained by its type.

While it is possible to construct a secondary designator to an object defined with `const` so that the secondary designator does not have the `const` qualifier, any attempt to modify an object through such a designator still has undefined behavior ([II99, 6.7.3§5], [II11b, 6.7.4§6]).

F.8 The fine print in the C semantics

- Any expression expressed in C is a designator for an object; however while some expressions are secondary designators for an object that already exists, some other

expressions are primary designators for object defined through them (e.g. intermediary computation results). Which expressions are primary designators and secondary designators is entirely specified.

- The C syntax does not allow a program to designate an array item object without taking the address of the array. Due this feature, an array object which cannot have its address taken (like when `register` is used) cannot be used in any useful way.
- If an object cannot have its address taken, then:
 - the only kind of secondary designator that can be constructed for the object are (compatible) cast expressions and union and structure field accesses; in particular, array subscript and automatic conversion of arrays to pointers are not possible (by definition) for arrays that cannot have their address taken;
 - it is always possible to identify the object, its primary designator (if any), its lifetime and its other properties (mutability, size) from any of its secondary designators during compilation;
 - the visibility of any secondary designator to the same object is always a subset of the object’s lifetime;
 - it is possible to identify all its secondary designators during compilation.

These properties extend to any object, in a given program, that *is proven, within that program, to not have its address taken* (even if it could be taken). This says that if the property “a specific object does not have its address taken” is verified to be true for a given object in a program up to a specific point, then the properties above hold for that object up to the same specific point.

- Non-array function parameter declarations are primary designators to objects whose initial value is a copy of the value of the object designated by the argument in the caller, whereas array function parameter declarations are secondary designators to the object designated by the argument in the caller.
- While the qualifiers on a secondary designator constrain what operations can be expressed in the language using that designator (for example, a designator with the `const` qualifier cannot be used for a write operation), they do not have any impact on the essential mutability of an object. If the object is mutable, then at least one of the following always hold:
 - the non-const qualified primary designator is visible (then it can be used to write to the object),
 - a new non-const qualified secondary designator can be constructed from a const-qualified secondary designator and used for write operations with the usual operational semantics of writes.

In particular, the program in listing F.1 is valid.

```
1 void foo(const int a[]) {
2
3 // here ‘a’ is a const-qualified secondary
4 // designator for the array object given as
5 // argument by the caller.
6
7 // so we can construct a new secondary
8 // designator to drop the const qualifier
9 // and change the type.
10 char *p = (char*) &a;
11
12 // the following is always valid if
13 // the array in the caller has at least
14 // one item and is mutable.
15 a[0] = 10;
16 }
17
18 int main(void) {
19     int a[10];
20     foo(a); // valid call, first byte of
21             // array is modified
22     return 0;
23 }
```

Listing F.1: Const designator to a mutable object.

Appendix G

Original language interface

—Description and defects

Abstract

Prior to our work some research had been realized to design a C-based interface to the proposed architecture. This appendix reviews this previous work and provides a comprehensive description of the corresponding language design. Through analysis, we uncover hidden complexity in this prior language design, and provide a proof that the language cannot be compiled to some variants of the target architecture interface from chapter 4.

Contents

G.1	Analysis of pre-existing sources	278
G.2	Hidden complexity	284
G.3	Fundamental problem	289
G.4	Summary and conclusion	292

Program	Description
<code>fibonacci</code>	Prints the first values of the Fibonacci sequence (the number of values is statically configured).
<code>frame-invert</code>	Computes the “negative” of a 2D, 8bpp black picture represented as an Iliffe vector [Ili61] (a one-dimensional array of pointers to one-dimensional arrays of pixel for each row in the image) of static size, and prints the result.
<code>matmul</code>	Computes the matrix-matrix product of static square integer matrices (static size and values), and prints the result.
<code>sine</code>	Computes the sine of a floating-point value using a 9-level Taylor expansion, and prints the result.

Table G.1: Example programs using the proposed C extensions, December 2008

Key	Title
[Jes06a]	μ TC - an intermediate language for programming chip multiprocessors.
[Jes06b]	Microthreading, a model for distributed instruction-level concurrency.
[BJK07]	Strategies for compiling μ TC to novel chip multiprocessors.

Table G.2: Academic publications related to the C language extensions, December 2008

G.1 Analysis of pre-existing sources

The process to re-construct the prior art was bootstrapped by a preliminary analysis of what information was already available. The following is an exhaustive list of the sources we found relevant to the language extensions at that point (late 2008):

- <i> a library of 4 short example programs written using the proposed primitives, listed in table G.1;
- <ii> 4 academic publications containing descriptions of the language constructs and examples, listed in table G.2;
- <iii> an internal (unpublished) technical report describing the language extensions [LB08];
- <iv> program source code for “`utc2cpp`,” a translator from the proposed language to a software-based concurrency run-time system [vTJLP09] with different semantics from the proposed hardware architecture;
- <v> the various authors of the previous sources themselves.

During this research, we also found 7 additional example programs using the proposed language extensions, developed in a separate project (*ÆTHER*). However, since these programs use additional concurrency semantics that are not supported by the proposed architecture (chapters 3 and 4), they are not further considered here.

G.1.1 Previously consistent statements

Close study of the aforementioned sources reveal the following consensus throughout:

- a new language construct based on the word “`create`” is added to the C language, with a syntactic structure similar to C’s `for` construct, and which can appear in C code in the same syntactic context as other *statements*;

```

1 #define N 100
2 int A[N], B[N];
3
4 thread scale1(void) {
5     index int i;
6     B[i] = A[i] * 2;
7 }
8
9 ...
10 {
11     int fl;
12     create(fl; 0; N; 1; 0) scale1();
13     sync(fl);
14 }
```

Listing G.1: Example code using the “create” construct and separate thread function to scale a vector.

- the “create” constructs describes a concurrent computation composed of multiple instances, called *logical threads*, of a *thread program* described by a C *syntactic block* (text between matched curly braces { and });
- the overall structure of the new construct is the word “create,” followed by *concurrency parameters* between parentheses, followed by a description of the work to do;
- the thread program can be specified as a syntactic construct analog to a C *function definition*, but including the special word “thread” in the function declarator;
- the concurrency parameters include the *start*, *limit* and *step* values that determine the number of logical threads and the logical thread index range (cf. section 4.3.2.3), as well as a *block* value that constrains the maximum breadth of actual parallelism (cf. section 4.3.2.2);
- in the program description of the basic computation unit, the word “index” can be added to an integer variable declaration to signify that this variable will be automatically initialized by the processor to the logical thread index;
- in the surrounding syntactic block, the control flow is synchronized with the completion of the concurrent computation at the point where the word “sync” appears;
- the occurrences of the words “create” and “sync” are matched together in the program code by the use of a common lexical identifier in the surrounding program text, which is to be assigned a “*family identifier*” value upon the creation of the concurrent computation;
- the unit of work described by the composition of “create,” “sync” and the related syntax can be replaced by a for loop in C without changing the computation semantics of the program;
- the special value 0, when specified for the *block* parameter, indicates that the underlying architecture is free to select the effective amount of parallelism when instantiating the work;
- each thread program can itself contain uses of “create” and “sync,” to allow the hierarchical composition of multiple levels of concurrent computations at run-time.

An example code that illustrates these statements is given in listing G.1. In this example, two static arrays A and B are defined in the global scope. Then a thread program

```

1 thread innerprod(int* X, int* Y, shared int s) {
2   index int i;
3   s = X[i] * Y[i] + s;
4 }

```

Listing G.2: Example thread program using a channel interface specification.

that describes one instance of a concurrent computation is defined using the word “**thread**,” using also “**index**” to declare the discriminating logical index *i*. This thread program can then be *used* in another computation block with “**create**” followed by the concurrency parameters between parentheses, followed by the name of the thread program. Then the word “**sync**” indicates that the control flow of the surrounding block should be suspended until the concurrent parallel computation completes (bulk synchronization). The concurrency parameters are specified by *positional parameters*, that is, their order specifies their role: the *start* value appears first, followed by the *limit*, *step* and *block* parameters. The name of the variable assigned the family identifier is listed as first parameter for both “**create**” and “**sync**.”

In this example, the basic thread program “scale1” only assumes that the index variable is automatically initialized and that a memory interface (load, store) to the arrays A and B is available. In particular, the base addresses of the arrays A and B are not a dynamic input to the basic computation: the position of the array definitions in the global scope imply that the linker program will rename statically all uses of the names “A” and “B” in the code to constant array locations in memory.

To define additional input and output channels to a basic thread program, the proposed language extension allows programmers to declare *thread parameters* in the definition of a thread program. All the sources isolated above agree on the following consensus:

- when used with the word “**thread**,” the syntactic position of function parameter declarations in the function definition (after the function name, between parentheses) is used to define the *interface* of the thread program;
- each position of the interface specification declares either one or two separate communication endpoint for the computation;
- at each position of the interface specification, the *name* used in the declarator will subsequently refer to the corresponding communication endpoints;
- at each position of the interface specification, the word “**shared**” can be added to distinguish declarations of “global” channels (where all concurrent instances of the computation must share the same source) from “shared” channels (where the endpoints are daisy-chained from one instance to the next). This is intended to correspond to the semantics of the hardware machine interface (cf. section 4.3.3);
- in the thread program, occurrences of the channel endpoint names are intended to be translated to *input* operations (for “global” channels) and either *input or output* operations (for “shared” channels), depending on whether the name is used as read-only operand of another C construct, or as the target of an assignment.

Remarkably, any channel declaration with the word “**shared**” simultaneously declares two channel endpoints, one for inputs and another, distinct one for outputs.

An example is given in listing G.2. In this example, the definition of the thread program “innerprod” declares 4 channel endpoints. The first two, designated by the names “X” and

“Y,” correspond to “global” channels in the underlying architecture, i.e. the assumption that all concurrent instances of the computation unit will share the same data source. The third position, where the word “**shared**” is used, simultaneously declares two endpoints with the common name “s.” The input part is used when the word “s” is subsequently used as an input operand in an expression, whereas the output part is used when the word “s” is used as the target of an assignment. The only statement in the program block can be decomposed as follows:

1. read the first array address from channel “X”;
2. compute the array element address “X[i]” using the logical thread index;
3. issue a load from memory using the address “X[i]”;
4. read the second array address from channel “Y”;
5. compute the array element address “Y[i]”;
6. issue a load from memory using the address “Y[i]”;
7. wait on completion of both memory loads, and add the results;
8. read from the input channel corresponding to the name “s”;
9. add the value read to the intermediate sum;
10. write the result to the output channel corresponding to the name “s”.

Finally, the design called for a means to relate the channel endpoints of the thread program to the data items in the surrounding code at the point of use of the “**create**” construct. This was done as follows:

- in the “**create**” construct, after the specification of which thread program to run, the program can specify a list of positional *thread arguments* with the a syntax similar to function calls: a comma-separated list of arguments enclosed in parentheses;
- each positional argument corresponds one-to-one to the endpoints of the channel interface specified in the thread program definition;
- for each position corresponding to a “global” channel, the program can specify any value, to be subsequently propagated by the substrate architecture as the source of input operations for all concurrent instances of the computation unit;
- for each position corresponding to a “shared” channel, the program can specify a *lexical identifier* referring to a C variable declaration in the surrounding scope. The value of this variable at the point the “**create**” construct is reached during execution is to be propagated by the architecture as the source of input operations by the first concurrent instance of the computation unit; and the value output by the last instance is to be propagated as the new value of the C variable after the control flow passes the “**sync**” word.

An example is given in listing G.3, which complements listing G.2. In this program fragment, the “**create**” construct uses the 3 positions of the interface defined by the thread program “innerprod.” The first two positions, corresponding to the “global” channel declarations X and Y in the definition of “innerprod,” are provided the computed values &A[0] and &B[0], which describe the start addresses in memory of the arrays A and B, respectively. The third position, corresponding to channel “s” in “innerprod,” is provided the identifier “res” which names a variable declared in the same scope. The value of “res” at the point the “**create**” construct is reached is determined by the assignment that immediately precedes (0 in this case), and this is the value sent as input to the first instance of the computation. When the last instance completes, the value it writes to its output channel for “s” is then

```

1 #define N 100
2 int A[N], B[N];
3
4 ...
5 {
6     int res, fl;
7     ...
8     res = 0;
9     create(fl; 0; N; 1; 0) innerprod(&A[0], &B[0], res);
10    sync(fl);
11    printf("The_sum_is_%d\n", res);
12 }

```

Listing G.3: Example use of the “innerprod” thread program.

propagated back to “res” in the invocation context. In the example this value is then printed to reveal the result of the computation.

G.1.2 Committed but unsupported features

The description given in the previous section is sufficient to explain the program code in source <i>; moreover, it is consistent with all the code fragments given as illustrations in source <ii>.

In this section, we provide the *additional* statements about the language design that have been stated in sources <ii> to <iv> which were not directly illustrated by example programs. Because they were stated in writing and agreed upon, they were assumed to be part of the language specification by the research community. However these are also the point where the language design became inconsistent with the hardware design.

The first additional statement concerns the relationship between the words “create” and “sync” in the language syntax. While the first publications [Jes06a, Jes06b] and source <i> are silent on this topic, the paper [BJK07] and sources <iii> and <iv> indicate that:

- the construct starting with the word “create” and ending with the thread argument specification and a semicolon can be used in the syntactic place of a C statement, and
- the construct starting with the word “sync,” followed by a family identifier variable between parentheses and a semicolon, can also be used in the syntactic place of a statement, albeit at a *different statement position* in the program than the “create” construct.

In other words, this additional formulation *decouples* the “create” and “sync” part of a concurrent work definition. Although no example was given alongside this statement, its consequences are clear: the design allows the expression of other statements, control flow structures or blocks in the program text between the “create” and “sync” parts. The motivation of this clarification was twofold. The first was increase the amount of concurrency that could be expressed, by allowing the description of extra computations that could be carried out simultaneously with the concurrent work described by “create,” before the “sync” construct is reached by the control flow. However, the main motivation for this extra step was to also state that:

Side note G.1: Attempt to constrain the well-formedness of programs.

Interestingly, the last part of the quoted statement, which tries to establish a criterion for the well-formedness of programs, actually fails to do so. Indeed, for any *actual* execution of a thread program containing both “**create**” and “**sync**” constructs, the execution can be described as a *single* sequence of intermediate steps which describe how the control flow *actually* passes all sequence points in the program¹. In any such sequence, observed at the point in time “**sync**” occurs, at most one “**create**” antecedent will have assigned a value to the same family identifier variable prior to that point. Because the execution of a thread program is (conceptually) sequential, there is no circumstance where more than one “**create**” produces the family identifier value eventually consumed by “**sync**.” Therefore, this last sentence, while logical, provides no sensical criterion to determine whether a program is well-formed or not.

- for any given “**create**” construct, the corresponding “**sync**” construct is optional; that is, a program can define an *asynchronous* computation which is not waited upon by the creating thread.

This extra step forward in the design was pressured by project partners during the separate project *ÆTHER*, as it was perceived to be required to implement a run-time system for the coordination language S-NET [GSS10, JS08], later revisited in [H10, Chap. 7]). The step was made and the corresponding language feature implemented in the software-based run-time system [vTJLP09] that was used as prototype before the Apple-CORE project started. However, this decision was taken independently from the architecture research that led to the design we describe in chapters 3 and 4.

Finally, in source <iii> (Section “Microthread C,” sub-section “**sync**”), we found the only further constraint on the relative placement of the constructs in the text of a program:

“The sync function must not be used outside of a thread function and the argument must identify a variable [...] defined in the scope of the thread function in which the sync occurs and must under all circumstances be the direct result of a single create statement in the same scope as the sync. That is, the program is malformed if there exists the possibility that the sync is used on [...] the result of different creates.”

The main message of this statement is that

- both “**create**” and “**sync**” constructs pertaining to the same variable holding a family identifier must appear in the same scope.

This statement further prevents a program to be structured in such a way that a concurrent work unit is created in one thread, and synchronized upon in a different thread, as this feature was not available in the underlying hardware architecture.

G.1.3 Overly restrictive statements

In source <iii> (Section “Microthread C,” subsection “Program Structure”), we further find that

¹This is opposed to the abstract view of the control flow graph as a whole, which considers simultaneously all possible paths through the graph.

```

1 void bar(int *p) {
2     *p = 10;
3 }
4
5 thread foo(shared int x) {
6     int *p = &x;
7     bar(p);
8 }

```

Listing G.4: Using the “address of” operator on a channel endpoint.

“There are no regular C function calls allowed in [the proposed language extension]. All [functions] must be defined as thread functions and ‘called’ via the create action.”

This conflicts with the requirements of generality set forth in section 6.2.1.

G.2 Hidden complexity in the proposal

To gain further understanding of the proposed language semantics, we can attempt to craft programs which exercise specifically what the documented prior work did *not* state about the new constructs. This revealed additional, *hidden complexity* in the design which was not envisioned by the research group previously. These are described here.

G.2.1 Nature of thread parameters

This code has no meaning with the target machine interface. Indeed, the communication channel endpoints are hardware synchronizers which cannot be indirectly addressed (section 4.2).

In more general terms, thread parameter names are not designators for C objects. As per the terminology introduced in Appendix F, the endpoints of communication channels do not have storage, so they do not fall under any of the categories for objects in C. Moreover, [II99, III1b] indicate that “an identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter.” By introducing a new concept which does not fall in any of these categories, the “thread parameter,” the proposal thus requires to extend the definition of *identifiers* in C and the semantics of identifiers in expressions. We must first extend the following properties of identifiers:

- about *scope*, [II99, 6.2.1§2] and [III1b, 6.2.1§2]: thread parameter identifiers have *block scope*, over the extent of the thread program body;
- about *name spaces*, [II99, 6.2.3] and [III1b, 6.2.3]: thread parameter identifiers are *ordinary identifiers*, i.e. they are in the same name space as ordinary variables, base types, typedef names and enumeration constants.

Then we consider that C only allows identifiers that refer to objects and functions in expressions ([II99, 6.5.1§2] and [III1b, 6.5.1§2]). This needs to be extended; however we must be careful in doing so: it is not sufficient to specify that “thread parameter identifiers

are primary expressions,” we must also indicate *how they are converted to a value* that can be computed upon. To do this, we start by looking at the way C gives values to identifiers for objects and functions:

- 6.5.1§2 indicates that an identifier that designates an object is an lvalue; then 6.3.2.1§2 indicates an lvalue which does not have an array type is converted to the value stored in the designated object (and is no longer an lvalue); 6.3.2.1§3 indicates that an lvalue which designates an array is converted to a pointer to the start of the array (and is no longer an lvalue).
- 6.3.2.1§4 indicates that function designators, when used in expressions, are converted to a pointer to the function.

None of these statements apply to thread parameters; in particular thread parameters are not lvalues. Based on the sources we have isolated, we construct the following additional statement:

- identifiers for thread parameters are primary expressions; identifiers for “global” channels are converted to the value read from the corresponding channel endpoint; identifiers for “shared” channels are converted to the value read from the input endpoint of the corresponding channel pair; in both cases they are not lvalues.

With this statement, an expression of the form “ $x + 3$ ” where “ x ” denotes a thread parameter becomes meaningful and has a value. Meanwhile, an expression of the form “ $\&x$ ” is disallowed, because “ x ” is converted to a non-lvalue first, so the $\&$ operator cannot be applied any more.

However, this is not sufficient to provide a meaning to all examples shown so far. Indeed, since thread parameter identifiers are not lvalues, they cannot appear at the left hand side of assignment operators (“ $=$ ” “ $+=$ ” etc., as per [II99, 6.5.16§2] and [III1b, 6.5.16§2]). A new meaning is thus required to allow expressions of the form “ $x = x + 3$ ” where “ x ” denotes a thread parameter.

Most of the extra complexity lies here: the entire semantics of assignments in C require an object on the left-hand side of the operator. The constraints about typing, the semantics of the assignments regarding object representation and overlap, etc., are defined with the assumption that the target of the assignment is an object. To provide meaning to the examples, we must extend the definition of assignments to signify the output of a value on the outgoing endpoint of “shared” channels.

Moreover, the main definition of assignments ([II99, 6.5.16§3] and [III1b, 6.5.16§3]) indicates that the value of the entire assignment is the value of the left operand after the assignment. This is meaningless in our setting, and even quite dangerous, since the input endpoint of a “shared” channel may not be readable any more after the output endpoint has been written to (side note G.2).

In order to address this situation, the only way we found to give a meaning to the examples that would be otherwise compatible with existing C semantics is to create a *new statement* dedicated to the outgoing communication over “shared” channels:

- Syntax: an identifier that designates a thread parameter for a “shared” channel, followed by the assignment operator, followed by an expression, followed by a semicolon, is a *shared channel assignment*; this extends the syntax of expression statements ([II99, 6.8.3] and [III1b, 6.8.3]):

Side note G.2: About the input availability of “shareds” after writes.

With the “hanging” synchronizer mapping introduced in section 4.3.3.3, the endpoints of the outgoing “shared” channels to the first thread and the incoming “shared” channels from the last logical thread in a family are mapped to the same synchronizers in storage. This means that any value populated for the first created thread is overwritten by outputs performed by the last thread. If there is only one thread in the created family, this implies that the input endpoint in the created thread cannot be read reliably after the output endpoint is written to (the read will not produce the original value).

expression-statement:

identifier = *expression* ;
*expression*_{opt} ;

This definition introduces a syntactic ambiguity between the new shared channel assignment, and assignment expressions where the left hand side lvalue is an object identifier. However the ambiguity can be resolved by disambiguating identifiers during lexical analysis, by ensuring that thread parameter identifiers have their own lexical class. No conceptual difficulty is added here, because lexical disambiguation of identifiers is already a prerequisite of C (e.g. to disambiguate “x * y;” which can be either an expression if the first identifier refers to an object, or a declaration if the first identifier refers to a type).

- Semantics: a shared channel assignments emits the value of the expression on its right hand side to the outgoing channel endpoint of the “shared” channel pair designated by its left hand side.

We can then simply extend this definition to compound assignment operators (“+=”, etc) by establishing an equivalence between the form “x += E ;” and the form “x = x + E ;”

To summarize, we found that proper handling of thread parameters requires a new abstract concept in the C language, an extension of the definition of *identifiers*, and a new *statement* dedicated to communicate through the outgoing channel of a “shared” channel pair. This was not visible in the original language extension proposal.

G.2.2 Addressability of thread arguments

Consider the example fragment in listing G.5. This fragment appears valid. The “create” construct defines a concurrent unit of work using the thread program “foo.” It also properly uses a local variable identifier as a thread argument for the “shared” channel, which provides the declared initial value of “x” (5) as the source value for the computation. Later in the program code the “sync” construct is used with the valid family identifier variable to wait on termination of the concurrent computation, and the final value of “x” is only assigned to “y” after the control flow passes “sync.”

However, what is the value of variable “y” after execution passes the sequence point on line 14? How to ensure this?

According to the object semantics of the C language, detailed in Appendix F:

- line 7 defines an object of type “int” designated by the name “x;”
- line 9 defines an object of type “int*” designated by the name “p;” and
- it also defines the expression “*p” to be a secondary designator for the object designated by “x.”

```

1  int offset = 0;
2  int* identity(int* p)
3  { return p + offset; }
4  thread foo(shared int x) { ... }
5
6  void bar(void) {
7      int x = 5;
8      int f, y;
9      int *p = &x;
10     p = identity(p);
11
12     create(f; 0; 1; 1; 0) foo(x);
13     sync(f);
14     y = *p;
15 }

```

Listing G.5: Pointer aliasing a channel endpoint.

From that point onward, C mandates that “x” and “*p” refer to the same object, as well as any expression of the form “*E” where the pointer value E is equal *at run-time* to the value of the expression “&x.” Since expressions of the form “*E,” when the value of E is not statically known, are always handled as memory loads, this implies that the object designed by both “x” and “*p” must be stored in memory, at an address copied to the pointer object designed by “p.” This applies here because the value of the object designed by “p” cannot be statically determined after line 10 (since the global-scope variable “offset” *may* have a non-zero value at run-time, it is not possible to derive statically the certainty that the function named “identity” always returns its argument).

However, when execution reaches the “create” construct, the value of the object designed by “x” *must* be present in machine register so it can become the source value for the input communication in the thread program “foo” (section 4.3.3.3) This appears at first sight to conflict with the declaration semantics described above; to resolve this conflict a compiler must duplicate the object into a register upon the “create” construct, reserve the register until the “sync” construct, and copy back the value from the register to the same memory location immediately after the “sync” construct so that the subsequent use of “*p” produces the desired behavior.

This entails extra complexity in the implementation because existing C compilers internally only associate one storage for each object at each point in the control flow graph (either register or memory). Here the memory location must be preserved, alongside the register name used as channel endpoint, along the edges of the control flow at all points between the “create” and “sync” constructs.

G.2.3 Two-way implicit type conversions

Consider the example program fragment in listing G.6. In this fragment the name “z” designates an object of type “float.” However, the thread interface of “foo” expects a channel of type “int.” In the machine interface, different register classes are used for integer and floating-point values: the value read by the instance of “foo” *must* be present in an integer register. This appears to conflict with the declared type of “z.” To resolve this conflict, a

```

1 thread foo(shared int x) {
2     x = x / 2;
3 }
4 ...
5 {
6     int f;
7     float y, z = 4.5;
8     create(f; 0; 1; 1; 0) foo(z);
9     sync(f);
10    y = z;
11 }

```

Listing G.6: Implicit endpoint type conversion.

```

1 thread foo(shared int a,
2         shared int b, shared int c);
3 thread bar(shared int d,
4         shared int e, shared int f);
5 ...
6 {
7     int x = 1, y = 2, z = 3;
8     int f1, f2;
9     create(f1; 0; 1; 1; 0) foo(x, y, z);
10    create(f2; 0; 1; 1; 0) bar(z, y, x);
11    sync(f1);
12    sync(f2);
13 }

```

Listing G.7: Multiple orderings of the same endpoint names.

compiler must automatically reserve an integer register upon the “**create**” construct and convert the floating-point value from the object designated by “z” to this register prior to the family creation. Then it must also convert back the final value of this register after the “**sync**” construct into the object designated by “z.”

This entails extra complexity in the implementation because existing C compilers only introduce automatic type conversions between sequence points for designated objects. Here the conversion must occur before the sequence point that immediately follows “**sync**”, but there is no object designator mentioned at point in the program text. To support this a compiler must track internally all objects that need conversion from the point where “**create**” is used until the point “**sync**” is used.

G.2.4 Reuse of “shared” thread arguments

Consider the fragment in listing G.7. This code is valid according all the definitions introduced so far. The difficulty with this example is to determine its semantics. The definitions up to this point open two possible views:

- in the first view, which we shall name “abstruse and broken” (A&B) hereafter, the variables designated by “x,” “y,” and “z” are established as *semantically synonymous*

with the communication channel endpoints. In this view, the first creation at line 9 has well-defined semantics, and the initial values for “x,” “y” and “z,” respectively 1, 2 and 3, are used as the source values for the first created family. Then, because the names are synonymous with the channel endpoints, their value becomes undefined immediately after the “`create`” construct: as soon as the first creation occurs, there is no scheduling guarantee and the created thread may concurrently output new values to either “x,” “y” or “z” before the second “`create`” construct is reached at line 10 in the creating thread. So the second creation has a race condition. Then another race condition exists at the first synchronization on line 11 since the second family *may* write to its final output “shared” channel before the first family does and terminates. This second race condition persists even if the synchronization order is inverted.

- in the second view, which we shall name “cumbersome and defective” (C&D) hereafter, the variables designated by “x,” “y,” and “z” have their own storage, are *copied* to channel “buffers” upon family creation, and copied back from the channel buffer to their own storage upon family synchronization. In this view, both creations have well-defined semantics, and the initial values for “x,” “y” and “z,” are used as source values for all “shared” channels “a” to “f.” Then the values of the variables remain unchanged in the creating thread until line 11 is passed, at which point they are updated with whichever values were produced by the first family; then they are updated again at line 12 with the values produced by the second family.

More generally, with the view A&B, the semantics become undefined due to race conditions as soon as a common variable name is used as “shared” thread argument for two families created from the same thread concurrently. This race condition is resolved in view C&D, at the expense of doubling the space requirement and extra copy operations at each creation and synchronization. The extra space requirement is especially troublesome, because it must be allocated from the register name space which is typically small (32/64 registers). Also, these extra costs defeat a primary benefit of the architecture, namely *lean* and *fast* family creation and synchronization.

G.2.5 Summary of the hidden complexity

We do not detail further the implementation complexity, in a compiler, of either of these views, because we eventually side-stepped this issue entirely in the solution presented in chapter 6. However, we can summarize this section as follows: were the proposed language extensions adopted, this examples exposes yet another unforeseen complexity in the design: either broken semantics, or cumbersome extra channel buffers.

G.3 Fundamental problem with the original language proposal

In this section, we prove that the language described in this appendix, in contrast to the one described in chapter 6 and Appendix I, cannot be compiled to a machine interface with “fused” creation such as described in section 4.3.1.2.

G.3.1 Non-composability within statement blocks

Let us consider the program fragment in listing G.8. This fragment is correct according to the original proposed language semantics described above. Its control flow is given in fig. G.1.

```

1 thread foo(shared int x);
2 int test(int x);
3
4 void bar(int N) {
5     int i, x, f;
6     for (i = 0; i < N; ++i) {
7         x = i;
8         create(f; 0; 1; 1; 0) foo(x);
9         if (test(f) == 0) {
10            sync(f);
11            i = x;
12        }
13    }
14 }

```

Listing G.8: Example program fragment using “create”

When control enters the body of the function “bar,” it reaches a loop. At each iteration, a concurrent work unit is created with one logical thread, given the iteration counter as input value for a “shared” channel. The family identifier value produced by the “create” construct is then provided as argument to the function “test,” and depending on whether the call to “test” evaluates to 0, the family is synchronized upon before the next iteration starts and the iteration counter reset with the output value of the family’s “shared” channel. The important property of this example is that the number of loop iterations, and whether the family is synchronized upon, is decided at run-time: neither the value of “N” nor the return value of “test” can be proven statically. Moreover, it is not possible to reorder the loop body, because there is a data dependency between “create” and the call to “test,” a control flow dependency between the call to “test” and the use of “sync,” and a data flow dependency between each use of “sync,” and the next loop iteration. Finally, if the thread program “foo” is known to be deterministic, and “test” is known to always returns 0 at run-time, all executions of the function “bar” will always be fully deterministic (in particular without concurrent race conditions) regardless of the choice of semantics opened in Appendix G.2.4.

Although the program fragment has well-defined semantics, it *cannot* be compiled to the target machine interface using “fused” create operations (section 4.3.1.2).

To prove this, we start by reminding ourselves of the semantics of the fused creation: this binds a machine register name from the point of family creation until the family termination *at run-time*. To cause the thread execution to wait until termination of the created family (the effect of running through “sync” in the control flow), it *suffices* that execution reaches an operation that uses this register name as operand. Conversely, in order to *avoid* synchronization on termination (the effect of *not* running through “sync” in the control flow), it is *necessary* that all operations executed after the “create” instruction avoid using this register name as operand *until the thread terminates*, i.e. here until execution exits the control flow graph of “bar.”

Then we consider all potential encodings of basic block BB2 in fig. G.1. For any potential valid encoding E of this basic block, E necessarily contains an occurrence of the “create” instruction, since the “create” construct is present in the source language. Let us consider the register name “\$R” used as operand to the “create” instruction in E . To support the run-time case where edge E3 is taken and the family termination is not synchronized upon by

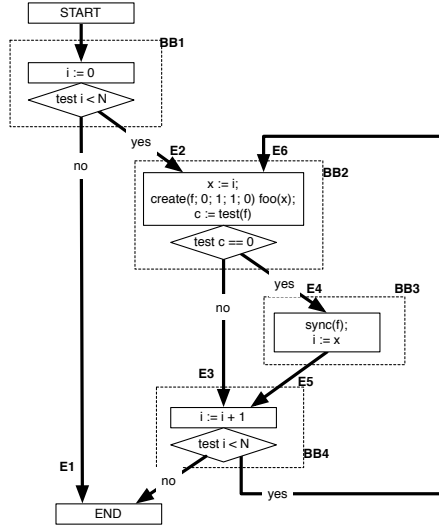


Figure G.1: Control flow graph of the function “bar” in listing G.8

the thread, necessarily *all further basic blocks down the control flow graph must avoid using “\$R.”* This includes basic block BB4, but also basic block BB2 due to the backward edge E6. Therefore E must not use “\$R,” which contradicts the definition of “\$R.” The contradiction entails that there exists no valid encoding of basic block BB2, i.e. that listing G.8 cannot be compiled to the machine interface.

This example is sufficient to prove that the source language cannot be compiled to a machine interface with fused creation, since there exists some valid input programs without a valid target encoding. When we initially proposed this proof, we were told that the example is convoluted, and that the input language definition might still be “good enough” if programs without cycles in the control flow graph can be compiled. Unfortunately, this is not possible either.

To understand why, let us consider the program fragment in listing G.9. This fragment contains a linear sequence of N uses of the “create” construct, where N is equal or greater the number of hardware register names M available in the target machine interface (e.g. $M = 32$, $N = 33$). Again, this program fragment is valid according to the original proposed language semantics. Let us thus consider any potential encoding E of this fragment. As seen above, the encoding of each “create” construct must avoid using any register name used by all previous “create” constructs that dominate it in the control flow graph. For the first “create” construct, one of M register names is used in E . For the second “create” construct, one of $M - 1$ register names is used in E . And so on inductively, until the M th “create” construct, where one in 0 remaining register names is used in E . Since this is not possible, E does not exist, i.e. listing G.9 cannot be compiled.

More generally, any potential compilation scheme for the proposed “create” construct, which can be used wherever a C statement can be used, does not compose under the sequential composition of C statements.

```

1  thread foo(void);
2  thread bar(shared int x);
3
4  void baz(void) {
5    int f1, f2, ... f⟨N⟩;
6    int x1, x2, ... x⟨N-1⟩;
7    create(f1;0;1;1;0) foo();
8    x1 = f1;
9    create(f2;0;1;1;0) bar(x1);
10   x2 = f2;
11   ...
12   create(f⟨N-1⟩;0;1;1;0) bar(x⟨N-2⟩);
13   x⟨N-1⟩ = f⟨N-1⟩;
14   create(f⟨N⟩;0;1;1;0) bar(x⟨N-1⟩);
15   sync(f⟨N⟩);
16   if (x⟨N-1⟩) {
17     sync(f1);
18     sync(f2);
19     ...
20     sync(f⟨N-1⟩);
21   }
22 }
```

Listing G.9: Example program fragment using the “**create**” construct.

G.3.2 Soundness argument

When exposed with the argument above, the designers of the original proposal suggested to further restrict usage of the “**create**” construct in specific programs depending on *whether register allocation succeeds for the surrounding thread program*; that is, advertise to users of the compiler technology that the *validity* of source code can only be checked by *actually* trying to process the source code through the *specific implementation* of register allocation in a core compiler technology.

This strategy amounts to defining a programming language that is *unsound*: if two program fragments A and B are valid because they can be compiled to their machine representation $[A]$ and $[B]$, it does not entail that the semantically valid sequential composition “ $A; B$ ” can be compiled to a machine representation $[A; B]$.

Since we should deem soundness a strongly desirable property of any machine interface to a general-purpose processor architecture, an opinion shared throughout our research community, we conclude that the proposed interface language was *inappropriate* for use with the proposed architecture.

G.4 Summary and conclusion

In this appendix, we have thoroughly analyzed prior work on designing a C-based interface to hardware microthreading. Our analysis shows shortcomings of the prior language design, including the impossibility to compile the proposed language constructs to a hardware implementation offering “fused” thread creation as proposed in section 4.3.1.2. This conclusion justifies why we did not reuse and built upon this prior work.

Appendix H

“Quick and dirty” compilation

—Massaging existing technology as a practical approach to code generation

Abstract

This appendix complements chapter 6 and details how we implemented code generation from source code to the new target architecture. To achieve this, we encapsulated an *existing* C compiler *unchanged* between a context-free source-to-source transformer and a post-processor on the assembly source. We then encapsulated the entire compilation pipeline within a new user-facing command, which we call “`s1c`,” with command-line semantics similar to GNU CC’s “`gcc`” driver.

Contents

H.1	Thread programs and “global” channels	294
H.2	Multiple channel endpoints	298
H.3	“Shared” channel endpoints	301
H.4	Bulk creation	302
H.5	Sequential schedule	309
H.6	Floating-point channels	309
H.7	Declarations & separate compilation	312
H.8	Extending bulk creation	312
H.9	Support for function calls	312
H.10	Resulting compilation chain	315

```

1  .globl foo
2  .ent foo
3  .registers 1 0 3 0 0 0
4  foo:
5  ldpc $12
6  ldah $12,0($12) !gpdisp!1
7  lda $12,0($12) !gpdisp!1 # $12 := GP
8  ldq $11,a($12) !literal # $11 := &a
9  ldq $11,0($11) # $11 := a
10 swch
11 s4addq $10,$11,$10 # $10 := &(a[i])
12 swch
13 stl $g0,0($10) # a[i] := x
14 end
15 .end foo

```

Listing H.1: Hand-crafted thread program.

```

1 extern int * a;
2 void foo(int i, int x) {
3     a[i] = x;
4 }

```

Listing H.2: Hand-crafted C code.

H.1 Thread programs and “global” channels

We started by observing that besides the machine interface features directly related to multithreading, the machine code executed by a logical thread is identical to the machine code that would be executed by a procedure on a legacy architecture with the same ISA substrate. We thus conjectured that an existing code generator for the underlying ISA would be able to generate valid code for the “body” of thread programs on our architecture.

To test this conjecture, we considered random C patterns, then hand-crafted their valid representation as a thread program in assembly code. Separately, we compiled the C code with a regular (legacy) C compiler towards the same ISA. Then we compared the results.

As an example, let us consider the assignment “ $a[i] = x$ ” where “ a ” is declared in the global scope, “ i ” is the logical thread index and “ x ” is a parameter. A valid enclosing thread program in assembly code is expressed in listing H.1. In this example, the “.registers” directive indicates 1 “global” channel endpoint and 3 private (local) registers, as per sections 4.3.3 and 4.8.1. The first instruction loads the program counter in register \$12. The next two instructions load the GP pointer into register \$12 (cf. [Foua] for information about GP-based addressing on the Alpha ISA). The next “ldq” instruction loads the address of pointer “ a ” using the GP pointer, and the next “ldq” instruction loads “ a ” itself. Then the array is accessed as expected, using the thread index preloaded in \$10 as per section 4.3.2.3. The “swch” and “end” annotations indicate a fetch switch hint and thread termination, respectively (cf. sections 4.4 and 4.8.1).

We then wrote the C code in listing H.2 and compiled it to Alpha assembly using the GNU C compiler. The result is shown in listing H.3. When comparing this result with

```

1  .globl foo
2  .ent foo
3  foo:
4  ldah $29,0($27) !gpdisp!1
5  lda $29,0($29) !gpdisp!1
6  ldq $1,a($29) !literal
7  ldq $1,0($1)
8  s4addq $16,$1,$1
9  stl $17,0($1)
10 ret $31,($26),1
11 .end foo

```

Listing H.3: Alpha assembly generated using GNU CC.

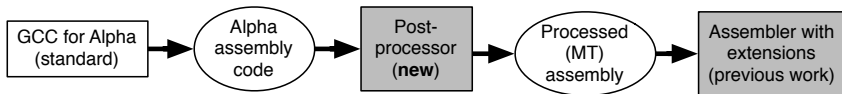


Figure H.1: Position of the assembly post-processor in the tool chain.

listing H.1, the following differences appear: the hand-crafted version contains “.registers” and “ldpc,” whereas the compiled version contains “ret” and a different use of register names. Otherwise, the instruction order and data flow are similar (the same in this example).

This similarity of structure persisted across most code fragments we exercised in this way. This allowed us to derive the minimal set of *transformations* needed to obtain the valid microthreaded assembly from the output of GNU CC:

1. insert “.registers” before the start of the function;
2. substitute the register names to match the structure of the thread register window;
3. insert “.ldpc” before the “ldah/lda” pair;
4. remove “ret” at the end, replace with “end” (end thread);
5. add “swch” where appropriate.

The 3 latter steps can be automated on the assembly text without interacting with the originating compiler. We did so by implementing a *post-processor*, inserted between the code generator from GNU CC and the assembler program, as depicted in fig. H.1. To add “swch,” we simply performed a definition-use analysis on the data flow of the assembly instruction operands, and placed “swch” after every instruction that *may* consume the output of a long-latency operation (e.g. the first instruction downstream of a memory load, FP operation, etc.)

Inserting “.registers” and translating the register names using a static translation table was possible in the post-processor as well, however this needs to know the desired thread interface. The thread interface is not expressed in the GNU CC output using the source in listing H.2. Also, we needed to ensure that the register name corresponding for the “global” channel endpoint would not be used by the code generator for other computations. Finally, we noticed that the “ret” instruction was often preceded by frame adjustments which are unnecessary in a thread program.

```

1 extern int * a;
2 sl_def(foo, int, x) {
3   sl_index(i);
4   a[i] = x;
5 }
6 sl_endif

```

Listing H.4: C code with strategically placed macro uses.

```

1 #define sl_index(IdxName) long IdxName = __mt_index
2 #define sl_def(FunName, Arg1Type, Arg1Name) \
3   void FunName(void) { \
4     register long __mt_index __asm__("$28"); \
5     register Arg1Type Arg1Name __asm__("$1"); \
6     __asm__volatile__(".registers_1_0_0_0_0_0"); \
7 #define sl_endif }

```

Listing H.5: Macro definitions for thread programs.

We automated the translation as follows. First we re-expressed the code from listing H.2 as depicted in listing H.4, hiding the function header and footer behind specially-named pre-processor macros. Then we implemented a helper file `sl_support.h` with the definitions from listing H.5. (Note that we use names prefixed with a double underscore “`__`” to void conflicts with existing identifiers in C code, cf. [II99, 7.1.3§1] and [III1b, 7.1.3].)

Then we needed to manage register renaming.

By investigating the GNU CC code generation back-end, we discovered that register allocation is performed in two passes: first all candidate variables are assigned to pseudo-registers, trying to minimize the number of pseudo-registers used, then the pseudo-registers are assigned to machine register names from a pool, *in a fixed order*. In other words, the implementation is such that a function that needs N different registers will always use the N first candidates in the allocation order. For this ISA target, the allocation order is 1, 2, 3, 4, 5, 6, 7, 8, 22, 23, 24, 25, 28, 0, 21, 20, 19, 18, 17, 16, 27, 9, 10, 11, 12, 13, 14, 26, 15 (REG_ALLOC_ORDER in `gcc/config/alpha/alpha.h`).

We used this newly gained knowledge as follows. First we excluded the first 12 registers of the allocation order using the command-line option “`-ffixed-$reg`” [Frec], in order to “make room” for inter-thread channel endpoints. We also compiled with “`-include sl_support.h`” to automatically prepend listing H.5 to the pre-processing unit:

```

alpha-linux-gnu-gcc -ffixed-$1 -ffixed-$2 \
-ffixed-$3 -ffixed-$4 -ffixed-$5 \
-ffixed-$6 -ffixed-$7 -ffixed-$8 \
-ffixed-$22 -ffixed-$23 -ffixed-$24 \
-ffixed-$25 \
-include sl_support.h -O2 test.c -S

```

The GNU CC output with these options is given in listing H.6. As desired, the name “`$1`” is only used in its role as a channel endpoint. The temporary variables are assigned

```

1  .globl foo
2  .ent foo
3  foo:
4  ldah $29,0($27) !gpdisp!1
5  lda $29,0($29) !gpdisp!1
6  .registers 1 0 0 0 0 0
7  ldq $0,a($29) !literal
8  ldq $0,0($0)
9  s4addq $28,$0,$28
10 stl $1,0($28)
11 ret $31,($26),1
12 .end foo

```

Listing H.6: Externally instrumented Alpha assembly.

Register name used by GNU CC	\$1	\$2	\$3	\$4	\$5	\$6
Standard alias/role	\$t0	\$t1	\$t2	\$t3	\$t4	\$t5
Substituted name	\$g0/\$d5	\$g1/\$s5	\$g2/\$d4	\$g3/\$s4	\$g4/\$d3	\$g5/\$s3
Register name used by GNU CC	\$7	\$8	\$22	\$23	\$24	\$25
Standard alias/role	\$t6	\$t7	\$t8	\$t9	\$t10	\$t11
Substituted name	\$g6/\$d2	\$g7/\$s2	\$g8/\$d1	\$g9/\$s1	\$g10/\$d0	\$g11/\$s0
Register name used by GNU CC	\$28	\$0	\$21	\$20	\$19	\$18
Standard alias/role	\$at	\$rv	\$a5	\$a4	\$a3	\$a2
Substituted name	\$l0	\$l1	\$l2	\$l3	\$l4	\$l5
Register name used by GNU CC	\$17	\$16	\$27	\$9	\$10	\$11
Standard alias/role	\$a1	\$a0	\$pv	\$s0	\$s1	\$s2
Substituted name	\$l6	\$l7	\$l14	\$l8	\$l9	\$l10
Register name used by GNU CC	\$12	\$13	\$14	\$26	\$15	\$29
Standard alias/role	\$s3	\$s4	\$s5	\$ra	\$fp	\$gp
Substituted name	\$l11	\$l12	\$l13	\$l15	\$l16	\$l17
Register name used by GNU CC	\$30	\$31				
Standard alias/role	\$sp	\$31				
Substituted name	\$l18	\$31				

Table H.1: Substitution table for register names.

to register names past the reserved range (“\$28”, “\$0”). The “.registers” directive is present, albeit not at the right position.

To refine the output, we then added the following steps in the post-processor:

1. substitute the register names using table H.1;
2. find the highest local register name *actually* used;
3. pull the inserted “.registers” directive at the top of the function definition and replace its 3rd parameter by the index of the highest local register name.

The result of these steps on listing H.6 is given in listing H.7. This is a valid thread program, which can be run on the new architecture.

However, some extra effort is required. As we can see in the edited assembly, as a special case in the allocation strategy outlined above, the special GP pointer is always assigned to “\$29.” The result of the substitution thus always uses the name “\$l17” if the original code needed GP, even though the thread program only needs 3 private registers for the computation. This is unsatisfactory, because the machine interface strongly suggests to

```

1  .globl foo
2  .ent foo
3  .registers 1 0 18 0 0 0
4  foo:
5  ldpc $l17
6  ldah $l17, 0($l17) !gpdisp!l
7  lda $l17, 0($l17) !gpdisp!l
8  ldq $l1,a($l17) !literal
9  swch
10 ldq $l1,0($l1)
11 s4addq $l0,$l1,$l0
12 swch
13 stl $g0,0($l0)
14 end

```

Listing H.7: Automatically edited Alpha assembly.

```

1  .globl foo
2  .ent foo
3  .registers 1 0 3 0 0 0
4  foo:
5  ldpc $l2
6  ldah $l2, 0($l2) !gpdisp!l
7  lda $l2, 0($l2) !gpdisp!l
8  ldq $l1,a($l2) !literal
9  swch
10 ldq $l1,0($l1)
11 s4addq $l0,$l1,$l0
12 swch
13 stl $g0,0($l0)
14 end

```

Listing H.8: Edited Alpha assembly with fewer used local registers.

reduce the number of effective private registers required by a thread program (sections 3.3.3 and 4.3.3). To address this, we add a “*compression*” algorithm to the post-processor. This algorithm iterates through all used local register names, and for each name it tries to find another name with a lower index which is not yet used. If such a name is found, the name is substituted throughout. This compression occurs before the register count in “.registers” is computed. With this algorithm, we obtain the final output in listing H.8. This is identical to the hand-crafted version from listing H.1: we successfully compiled a thread program, without changes to the substrate code generator in GNU CC.

H.2 Multiple channel endpoints

The previous mechanism relies on a C pre-processor macro to inject the desired thread interface, expressed by macro arguments, in the source code as variable declarations and the “.registers” directive. Since the C pre-processor cannot compute functions of the number of

```

1 #define sl_def1(FunName, Type1, Name1)
2   void FunName(void) { \
3     register long __mt_index __asm__("$28"); \
4     register Type1 Name1 __asm__("$1"); \
5     __asm__ volatile__(".registers_1_0_0_0_0_0");
6 #define sl_def2(FunName, T1, N1, T2, N2) \
7   void FunName(void) { \
8     register long __mt_index __asm__("$28"); \
9     register T1 N1 __asm__("$1"); \
10    register T2 N2 __asm__("$2"); \
11    __asm__ volatile__(".registers_2_0_0_0_0_0");
12 #define sl_def2(FunName, T1, N1, T2, N2, T3, N3) \
13   void FunName(void) { \
14     register long __mt_index __asm__("$28"); \
15     register T1 N1 __asm__("$1"); \
16     register T2 N2 __asm__("$2"); \
17     register T3 N3 __asm__("$3"); \
18     __asm__ volatile__(".registers_3_0_0_0_0_0");
19 /* ... and so on. */

```

Listing H.9: Macro definitions necessary for different interface arities.

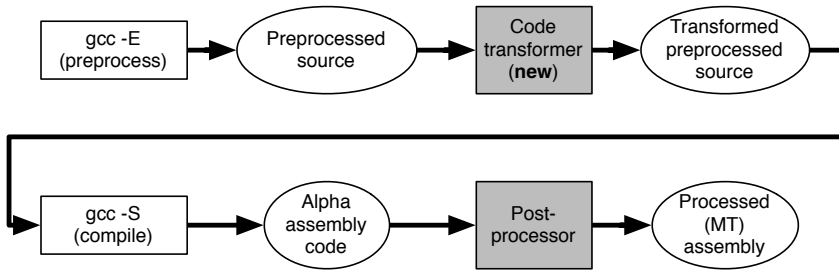


Figure H.2: Position of the code transformer in the tool chain.

macro arguments, this approach would require multiple macro definitions to support more than one channel endpoint, as in listing H.9.

While this approach is conceptually sound, it is technically impractical: for any set of macros the effective number of thread channels supported is limited by the largest macro definition. We considered instead that a dynamically computed expansion of the macro would be a more robust, future-proof approach. To achieve this, we inserted a call to M4 [KR77] as a *code transformer* between the *pre-processing* and *translation* phases of the C compiler, as depicted in fig. H.2, and we defined our “sl_” macros using M4.

While implementing the code transformer, we found that the implementation would be simplified if there was a syntactic unit around the pair “(type, name)” for each thread channel. We did so by extending the syntax definitions with a new form “sl_parm(*Type*, *Name*).”

With this setup, we are able to compile thread programs with multiple channel endpoints, e.g. listing H.10, to valid assembly output, e.g. listing H.11 (the comments on the right side of the generated assembly have been manually inserted to explain the result).

Side note H.1: About the avoidance of a C syntax parser.

Although we could have reused an existing comprehensive code transformation infrastructure able to understand the C syntax and semantics, we deliberately avoided doing so for the following practical reasons:

- overhead: the entry cost of learning about and mastering the existing tool seemed large given the relative simplicity of our approach;
- test cycles: the existing tools we found (e.g. CIL [NMRW02]) required separate build and execution steps to use; we thus deemed them inappropriate for the agile development style we wished to use;
- simplicity: since the syntax substitutions could be implemented in a context-free manner, any error could be easily traced to its origin by looking at the result of the substitution; with a structured tool instead, one would need to scan through a graph or tree of abstract representation to find the origin of errors.

As the story goes, we ended up implementing most the code transformer with Python [Foub] (instead of M4) to simplify the expression of the transformations. However, throughout our research the tool could perform context-free syntax substitutions. Our major goal, which was to avoid knowledge of the potential syntax extensions of the underlying C compiler, while keeping the ability to propagate them fully from input to output, was achieved this way.

Side note H.2: Preservation of line number markers in M4.

When adding M4 to the processing chain, we had to modify M4 to understand the line number markers added during the initial C pre-processing step (“`#line N`”). This is necessary because macro expansion in M4 may add or remove lines from the pre-processed text. Without extra support from M4, a discrepancy could appear between the line markers originally inserted by the C pre-processor and the source code after processing by M4, which would in turn render any subsequent error messages during compilation meaningless.

We performed this change in GNU M4. The change was minimal (less than 20 lines of code modified); it was submitted to the GNU M4 developers and is pending acceptance in the main GNU M4 distribution.

```

1  sl_def(scal , sl_parm(int* , b) ,
2      sl_parm(const int* , a) ,
3      sl_parm(int , c)) {
4      sl_index(i);
5      b[i] = a[i] * c;
6  }
7  sl_endif

```

Listing H.10: Code using multiple channel endpoints.

```

1      .globl scal
2      .ent scal
3      .registers 3 0 2 0 0 0 # 3 chans, 2 local regs
4  scal:
5      s4addq $10,0,$10 # $10 := index * 4
6      addq $g1,$10,$11 # $11 := a + index*4 (= &a[i])
7      addq $g0,$10,$10 # $10 := b + index*4 (= &b[i])
8      ldl $11,0($11) # $11 := a[i]
9      mull $11,$g2,$11 # $11 := a[i] * c
10     swch
11     stl $11,0($10) # store int to b[i]
12     swch
13     end
14     .end scal

```

Listing H.11: Generated assembly source for the “scal” thread program.

H.3 “Shared” channel endpoints

To implement “shared” channel endpoints (section 4.3.3.3), we could use register names from the same reserved block of 12 names. However, a new difficulty arised: there are now two endpoints per channel, one “incoming” and “outgoing.” Meanwhile, we wanted to keep the general look and feel of the original language proposal, and allow users to define a single name to designate both. We did this as follows. First we defined a new form “`sl_shparm(Type, Name)`” which populates the 2nd parameter to “`registers`” and generates two additional declarations at each use in the C function header, for example:

```
1 register Type __mti_Name __asm__("$24");
2 register Type __mto_Name __asm__("$25");
```

Then we defined two other forms as follows:

```
1 sl_getp(Name)
2 → (__mti_Name)
3 sl_setp(Name, Value)
4 → do { __mto_Name = (Value); } while(0);
```

(The use of “do...while(0)” ensures that the new construct can only be used in the syntax where a C statement is expected.)

While experimeting with this new form, we ran into two issues, related to the ability of the substrate GNU CC code generator to track aliases through assignments, which is otherwise a desirable optimization. The first issue is illustrated by the following sequence:

```
1 int t, u;
2 t = sl_getp(s);
3 sl_setp(s, t+1);
4 int u = t * t;
```

Without further attention, this would be translated to:

```
1 lda $t0, 1($s1) # $t0 := 1
2 addl $d0, $t0, $s0 # $s0 := $d0 + 1
3 mull $d0, $d0, $t0 # $t0 := $d0 * $d0
```

In other words, the code generator tracks that “t” is an alias for “__mti_*Name*” and reuses the same register name beyond “`sl_setp`.” This is incorrect in our setting, as explained in side note G.2. To address this, it suffices to indicate that any use of “`sl_setp`” also clobbers “__mti_*Name*,” to force “t” to alias a copy beyond that point. We do this as follows:

```
1 sl_setp(Name, Value)
2 → do {
3   __mto_Name = (Value);
4   __asm__ __volatile__("#nothing"
5     : "=r"(__mti_Name)
6     : "r"(__mto_Name));
7 } while(0);
```

As per [Frea] this syntax ensures that any prior alias to “__mti_*Name*” cannot use the original register name any more.

Then we experienced another issue: some C expressions would cause the target of an assignment to be assigned multiple times. For example:

```
1  sl_setp(s, x ? (x - 1) : 0);
```

would be compiled to:

```
1  subl $10,1,$s0 # $s0 := $10 - 1
2  cmoveq $10,0,$s0 # if $10=0 then $s0:=0
```

This is incorrect because each output to an outgoing channel is a synchronizing event and only the final result should be visible by the sibling thread. We addressed this by extending the definition of “`sl_setp`” as follows:

```
1  sl_setp(Name, Value)
2  → do {
3      typeof (__mto_Name) __tmp_set = (Value);
4      asm volatile ("mov_%4,%0"
5          : "=r" (__mto_Name),
6            "=r" (__mti_Name)
7            : "0" (__mto_Name),
8            "1" (__mti_Name),
9            "r" (__tmp_set));
10 } while (0);
```

With this construct, the value to output to the outgoing “shared” endpoint is always constructed in a local synchronizer before the final “`mov`” instruction effects the output. Note how we use “`typeof`” to define a temporary variable of the right type. Although we could have defined a parameter type lookup table in the code transformer, which would “remember” throughout a thread program body the type of each channel endpoint, this would fail to handle the following source:

```
1  typedef long mytype;
2  sl_def(foo, sl_shparm(mytype, x)) {
3      typedef char mytype[1000];
4      sl_setp(x, sl_getp(x)+1);
5  }
```

In this example the name “`mytype`” is changed to designate a different type within the function body, and thus after the function header. This new definition would be invisible to the code transformer because it appears outside of one of our constructs. If we would reuse the name “`mytype`” textually in the expansion of “`sl_setp`,” the code would become invalid. With “`typeof`” we ensure that the original definition type is used.

Finally, for symmetry, we then renamed the previous “`sl_parm`” construct for “global” channels to “`sl_glparm`” and we ensured that it declares variables of the form “`__mti_Name`” so that “`sl_getp`” can be used to access them.

With this in place, we are able to compile listing H.12 to listing H.13, which is a valid thread program for the new architecture.

H.4 Bulk creation

H.4.1 Code generation for “fused” creation

We cover first code generation for “fused” creation styles (section 4.3.1.2), as this was historically the first interface available.

```

1  sl_def(innerprod , sl_glparm(int *,a) ,
2      sl_glparm(int *,b) ,
3      sl_shparm(int ,sum))
4  {
5      int* a = sl_getp(a);
6      int* b = sl_getp(b);
7      sl_index(i);
8      sl_setp(sum, a[i] * b[i] + sl_getp(sum));
9  }
10 sl_enddef

```

Listing H.12: Source code for the “innerprod” thread program.

```

1  .globl innerprod
2  .ent innerprod
3  .registers 2 1 2 0 0 0
4  innerprod:
5  s4addq $10,0,$10 # $10:=i * 4
6  addq $g1,$10,$11 # $11:=&b[i]
7  swch
8  addq $g0,$10,$10 # $10:=&a[i]
9  swch
10 ld1 $11,0($11) # $11:=b[i]
11 ld1 $10,0($10) # $10:=a[i]
12 swch
13 mull $11,$10,$10 # $10:=a[i]*b[i]
14 swch
15 addl $10,$d0,$10 # $10:=a[i]*b[i]+$d0
16 swch
17 mov $10,$s0
18 end

```

Listing H.13: Generated assembly for the “innerprod” thread program.

As described in section 4.3.1, bulk creation is controlled at the machine interface via a sequence of “allocate,” “set” configuration operation and one of the “create” operations. To follow the look and feel of the original language proposal, we encapsulated this sequence behind a single construct, given in listing H.14.

We use the register constraint “rI” because the configuration instructions accept both an immediate constant or a register, and the code generator will choose the form that entails the least number of instructions. We use a counter in the code transformer to auto-generate a different value for N at every occurrence of “sl_createsync,” to avoid duplicate declarations of “__mt_fid.”

With this in place, we can compile listing H.15 to listing H.16.

Also, we use a condition in the code transformer to avoid emitting one of the configuration instruction if the corresponding parameter is omitted in the form “sl_createsync.” For example, our construct allows us to compile listing H.17 to listing H.18.

The next step concerns setting up communication endpoints with the created family. In the hardware implementations we have used, fused creation is always used with “hanging”

```

1 sl_createsync(Start, Limit, Step, Block, Fun)
2 →
3 long __mt_fidN;
4 asm volatile ("allocate_%0" : "=r"(__mt_fidN));
5 asm ("setstart_%0,%2"
6 : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Start));
7 asm ("setlimit_%0,%2"
8 : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Limit));
9 asm ("setstep_%0,%2"
10 : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Step));
11 asm ("setblock_%0,%2"
12 : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Block));
13 asm volatile ("crei_%0,%0(%1);_mov_%0,_$31"
14 : : "r"(__mt_fidN), "r"(Fun))

```

Listing H.14: Prototype “create” construct.

```

1 sl_def(foo) { } sl_enddef
2 sl_def(bar) {
3   sl_createsync(1, 2, 3, 4, foo);
4 } sl_enddef

```

Listing H.15: Example use of “createsync.”

synchronizer mappings (section 4.3.3.3). As per section 4.8.2, with such mappings the “setregs” instruction must specify static offsets in the creating thread’s register window where the endpoints must be set up. Also, these endpoints must contain the source values at the point the “create” instruction is executed. Finally, the endpoints are set up by using contiguous register names from the first offset indicated by “setregs.”

To implement this, we force the use of specific registers to pass arguments, as detailed in listing H.19. First we extend the interface of the “**sl_createsync**” form to accept a variable number of extra constructs of the form “**sl_glarg**(*Type*, *Name*, *Value*).” When G such constructs are provided, we then declare G local variables named accordingly, placing them in explicit registers. We choose the explicit register names by applying the *reverse substitution* from table H.1 to the new names “\$10...\$($G - 1$).” This produces the G first names from the sequence \$28, \$0, \$21, ... \$26, with the guarantee that the forward substitution will produce contiguous register names during post-processing.

We then expand the context allocation and family configuration as previously.

Then we assign the values specified in the “**sl_glarg**” forms to the new local variables. We place these assignment after the family configuration, instead of directly as initializers in the variable declarations, in order to preserve the order of side effects in the expressions passed to “**sl_createsync**.”

Finally, during creation, we force the new variables to be available in their respective registers at the point the “create” instruction is emitted. We construct the meta-syntactical variables “% N ” in the assembly pattern based on the number G of arguments that follow.

For “shared” channels, we extend as detailed in listing H.20. This is a straightforward extension of listing H.19, using the next S register names in the creating thread’s register

```

1  .globl foo
2  .ent foo
3  .registers 0 0 0 0 0 0
4  foo:
5  nop
6  end
7  .end foo
8  .globl bar
9  .ent bar
10 .registers 0 0 3 0 0 0
11 bar:
12 ldpc $12
13 ldah $12, 0($12) !gpdisp!1
14 lda $12, 0($12) !gpdisp!1 # $12 := GP
15 allocate $10 # $10 := context id
16 ldq $11, bar($12) !literal # $11 := &bar
17 setstart $10, 1 # set start := 1
18 swch
19 setlimit $10, 2 # set limit := 2
20 setstep $10, 3 # set step := 3
21 setblock $10, 4 # set block := 4
22 crei $10, 0($11) # create, $10 := future
23 swch
24 mov $10, $31 # wait on $10 (sync)
25 end
26 .end bar

```

Listing H.16: Generated assembly from listing H.15.

```

1 sl_def(foo,sl_glparm(int,x)) {
2   sl_createsync( , sl_getp(x), , , bar);
3 }

```

Listing H.17: Example use of “createsync” with omitted parameters.

```

1 allocate $10 # $10 := context id
2 ldq $11, bar($12) !literal # $11 := &bar
3 setlimit $10, $g0 # limit := 1st th. parm
4 swch
5 crei $10, 0($11) # create, $10 := future
6 swch
7 mov $10, $31 # wait on $10 (sync)

```

Listing H.18: Generated assembly for listing H.17.

```

1 sl_createsync(... Fun,
2             sl_glarg( $T_1$ ,  $N_1$ ,  $Val_1$ ), ... sl_glarg( $T_G$ ,  $N_G$ ,  $Val_G$ ))
3 →
4 long __mt_fidN;
5  $T_1$  __mta_ $N_1$  __asm__ ("$$RevSubst($l0)");
6 ...
7  $T_G$  __mta_ $N_G$  __asm__ ("$$RevSubst($l(G-1))")
8 /* ... allocate and configure happen here ... */
9 __mta_ $N_1$  = ( $Val_1$ );
10 ...
11 __mta_ $N_G$  = ( $Val_G$ );
12 __asm__ ("setregs_0,0,0,0,0"
13         : "=r"(__mt_fidN) : "0"(__mt_fidN));
14 __asm__ volatile ("crei_2G, 0(2G+1)"
15                 "\n\tmov_2G, $31"
16                 : "=r"(__mta_ $N_1$ ), ... "=r"(__mta_ $N_G$ ),
17                 : "r"(__mta_ $N_1$ ), ... "r"(__mta_ $N_G$ ),
18                 "r"(__mt_fidN), "r"(Fun))

```

Listing H.19: Support for “global” channel endpoints in “createsync.”

```

1 sl_createsync(... Fun,
2             sl_sharg( $T_1$ ,  $N_1$ ,  $Val_1$ ), ... sl_sharg( $T_S$ ,  $N_S$ ,  $Val_S$ ))
3 →
4 /* declaration for G variables here... */
5  $T_1$  __mta_ $N_1$  __asm__ ("$$RevSubst($lG)");
6 ...
7  $T_S$  __mta_ $N_S$  __asm__ ("$$RevSubst($lG+S-1)")
8 /* ... allocate and configure happen here ... */
9 /* ... also assignments to 1st G channel endpoints */
10 __mta_ $N_1$  = ( $Val_1$ );
11 ...
12 __mta_ $N_S$  = ( $Val_S$ );
13 __asm__ ("setregs_0,0,0,0,0"
14         : "=r"(__mt_fidN) : "0"(__mt_fidN));
15 __asm__ volatile ("crei_2(G+S), 0(2(G+S)+1)"
16                 "\n\tmov_2(G+S), $31"
17                 : /* G arguments here... */
18                 "=r"(__mta_ $N_1$ ), ... "=r"(__mta_ $N_S$ ),
19                 : /* G arguments here... */
20                 "r"(__mta_ $N_1$ ), ... "r"(__mta_ $N_S$ ),
21                 "r"(__mt_fidN), "r"(Fun))

```

Listing H.20: Extension of listing H.19 for “shared” channels.

```

1 #define N 100
2 int A[N];
3 int B[N];
4
5 sl_def(innerprod,
6     sl_glparm(int*, a),
7     sl_glparm(int*, b),
8     sl_shparm(int, sum)) {
9     int* a = sl_getp(a);
10    int* b = sl_getp(b);
11    sl_index(i);
12    sl_setp(sum,
13        a[i] * b[i] + sl_getp(sum));
14 } sl_enddef
15
16 sl_def(kernel, sl_shparm(int, res)) {
17     /* we pass A and B via thread
18     channels to avoid extra instruction
19     to load the array address in each
20     thread of the family. */
21     sl_createsync( , N, , , innerprod,
22         sl_glarg(int*, ,A),
23         sl_glarg(int*, ,B),
24         sl_sharg(int, sum, 0));
25     sl_setp(res, sl_geta(sum));
26 } sl_enddef

```

Listing H.21: Example vector-vector product kernel.

window for the endpoints of “shared” channels. We also add a new form “`sl_geta(Name)`” to retrieve the final value produced by the last thread in the family, as follows:

```

1 sl_geta(Name) → __mta_Name

```

With these constructs in place, we are able to compile listing H.21 to listing H.22. This example uses the SPARC ISA, as opposed to the Alpha ISA used in previous examples, to illustrate the generality of the approach; it is also one of the benchmarks that was eventually successfully run on the UTLEON3 prototype on FPGA. This implementation uses two machine instructions “setthread/create” instead of only one “crei,” but the semantics of the machine interface are otherwise identical.

H.4.2 Code generation with “detached” creation

We cover here the “detached” creation style from section 4.3.1.2.

Here code generation is implied, because the source values of the thread arguments do not need to be placed in specific (contiguous) registers any more. Instead, the “`sl_xarg`” constructs now simply expands to a use of the “puts” or “putg” operations for explicit communication of source channel values.

For bulk synchronization, we then expand to an explicit use of the new “sync” and “release” operations. The final values of “shared” communication channels are retrieved into local variables via “gets” between “sync” and “release.”

```

1   .global innerprod
2   .type   innerprod, #function
3   .registers      2 1 3 0 0 0
4 innerprod:
5   sll %t10, 2, %t11      ! %t11 := i*4
6   ld [%tg1+%t11], %t12  ! %t12 := b[i]
7   ld [%tg0+%t11], %t10  ! %t10 := a[i]
8   smul %t12, %t10, %t10 ! %t10 := a[i]*b[i]
9   swch
10  add %t10, %td0, %t10   ! %t10 := a[i]*b[i]+sum
11  swch
12  mov %t10, %ts0        ! %ts0 := a[i]*b[i]+sum
13  end
14  .size innerprod, .-innerprod
15  .global kernel
16  .type   kernel, #function
17  .registers      0 1 6 0 0 0
18 kernel:
19  allocate %t13          ! %t13 := context id
20  sethi %hi(A), %t10
21  or %t10, %lo(A), %t10  ! %t10 := &A[0] (1st gl.)
22  sethi %hi(B), %t11
23  or %t11, %lo(B), %t11 ! %t11 := &B[1] (2nd gl.)
24  mov 0, %t12           ! %t12 := 0 (1st sh.)
25  setlimit %t13, 100    ! set limit := 100
26  swch
27  sethi %hi(innerprod), %t15
28  or %t15, %lo(innerprod), %t15 ! %t15 := &innerprod
29  setthread %t13, %t15  ! set PC := &innerprod
30  create %t13, %t13     ! create, %t13 := future
31  swch
32  mov %t13, %t13        ! wait on $t13 (sync)
33  swch
34  mov %t12, %ts0        ! %ts0 := last sh. output
35  end
36  .size kernel, .-kernel

```

Listing H.22: Generated code for listing H.21, using the “fused” creation interface.

Detached creation is used in the hardware implementations together with the “separated” style of synchronizer mappings (section 4.3.3.3), so the restriction described in side note G.2 does not apply any more, which allows us to simplify the definition of “`sl_setp`” from Appendix H.3.

With this scheme in place, the translation of listing H.21 yields the code in listing H.23.

H.5 Sequential schedule

As explained in section 6.2.4, our technical approach must enable a sequential schedule of any construct expressing concurrency. With the constructs introduced so far, we build an equivalent sequential schedule as follows:

- thread program are replaced by C functions, taking a logical index value as 1st function parameter;
- family creation via “`sl_createsync`” is replaced by a loop;
- “global” channel endpoints are replaced by pass-by-value function parameters;
- “shared” channel endpoints are replaced by single pass-by-reference (pointer) function parameters.

The corresponding syntax expansions are provided in listing H.24. The pre-processor must keep an environment upon encountering “`sl_def`” in order to select the expansion of “`sl_getp`” in the function body (either direct or via pointer indirection); this is possible since we can implement data structures in the code transformer. As previously, the variable declarations at each use of “`sl_createsync`” must be disambiguated using a counter N increased at each occurrence of the construct.

H.6 Floating-point channels

Support for floating-point required extra attention because the register names for floating-point values and operations are different from integer register names.

While it was trivial to reuse the same principle for reserving register names away from register allocation, and establishing a substitution table based on the register allocation order in GNU CC (table H.2), integration in the language constructs mentioned so far ran into an obstacle: it is not possible to infer the *actual type* of a channel argument/parameter for the type name provided in the construct.

The specific question to be answered is: given an occurrence of “`sl_glparm(Type, Name)`” (or “`sl_shparm`”), should the expanded text use an integer or floating-point register name? The naive approach suggests looking at the syntactic form of the *Type* textual parameter; however this is inappropriate: if a C *typedef name* is provided ([II99, 6.7.7], [III1b, 6.7.8]), the actual type cannot be inferred without looking at the semantic context in the program. Since we wanted to avoid interfering with (or reimplementing) a C compiler front-end, we decided to sidestep the issue entirely and shadow all the relevant constructs with a floating-point variant: “`sl_glpfparm`” next to “`sl_glparm`,” “`sl_shfparm`” next to “`sl_shparm`,” and so on. The other constructs could remain unchanged.

With this extra support in place we were able to successfully compile code with floating-point channel types.

```

1  .global innerprod
2  .type    innerprod, #function
3  .registers    2 1 3 0 0 0
4  innerprod:
5  sll %t10, 2, %t11      ! %t11 := i*4
6  ld [%tg1+%t11], %t12  ! %t12 := b[i]
7  ld [%tg0+%t11], %t10  ! %t10 := a[i]
8  smul %t12, %t10, %t10 ! %t10 := a[i]*b[i]
9  swch
10 add %t10, %td0, %t10  ! %t10 := a[i]*b[i]+sum
11 swch
12 mov %t10, %ts0       ! %ts0 := a[i]*b[i]+sum
13 end
14 .size innerprod, .-innerprod
15 .global kernel
16 .type    kernel, #function
17 .registers    0 1 6 0 0 0
18 kernel:
19 mov 0, %t11          ! %t11 := 0
20 allocates %t10       ! %t13 := context id
21 sethi %hi(innerprod), %t12
22 or %t12, %lo(innerprod), %t12 ! %t12 := &innerprod
23 setlimit %t10, 100   ! set limit := 100
24 swch
25 crei %t12, %t10      ! create, %t10 := acknowledgement
26 sethi %hi(A), %t12
27 or %t12, %lo(A), %t12
28 putg %t12, %t10, 0   ! send &A to ‘‘global’’ 0
29 sethi %hi(B), %t12
30 or %t12, %lo(B), %t12
31 putg %t12, %t10, 1   ! send &B to ‘‘global’’ 1
32 puts %t11, %t10, 0   ! send 0 to first ‘‘shared’’ 0
33 sync %t10, %t11     ! sync, %t11 := future
34 mov %t11, %t11      ! wait on %t11
35 swch
36 gets %t10, 0, %t11   ! retrieve last ‘‘shared’’ 0
37 mov %t11, %t11      ! wait
38 swch
39 release %t10        ! release context
40 mov %t11, %ts0      ! propagate to next thread
41 end
42 .size kernel, .-kernel

```

Listing H.23: Generated code for listing H.21, using the “detached” creation interface.

```

1 sl_def(Name, sl_glparm( $T_G, N_G$ ), ... sl_shparm( $T_S, N_S$ ), ...)
2 →
3 void Name(long __mti,  $T_G N_G$ , ...  $T_S *N_S$ , ...)
4
5 sl_index(Idx) → long Idx = __mti
6
7 sl_getp( $N_G$ ) →  $N_G$ 
8 sl_getp( $N_S$ ) → ( $*N_S$ )
9
10 sl_setp( $N_S$ , Value) → do { ( $*N_S$ ) = (Value); } while(0)
11
12 sl_createsync(Start, Limit, Step, Block, Fun,
13     sl_glarg( $T_G, N_G, V_G$ ), ...
14     sl_sharg( $T_S, N_S, V_S$ ), ...)
15 →
16 long __mtiN, __mtbN = (Start), __mtlN = (Limit), __mtsN = (Step);
17  $T_G N_G$  = ( $V_G$ ); ...
18  $T_S N_S$  = ( $V_S$ ); ...
19 for (__mtiN = __mtbN; __mtiN < __mtlN; __mtiN += __mtsN)
20     Fun(__mtiN,  $N_G$ , ... & $N_S$ , ...)
21
22 sl_geta( $N_S$ ) →  $N_S$ 

```

Listing H.24: Syntax expansions for a sequential schedule.

Legacy register name	\$f10	\$f11	\$f12	\$f13	\$f14	\$f15
Standard alias/role	\$ft0	\$ft1	\$ft2	\$ft3	\$ft4	\$ft5
Substituted name	\$gf0/\$df5	\$gf1/\$sf5	\$gf2/\$df4	\$gf3/\$sf4	\$gf4/\$df3	\$gf5/\$sf3
Legacy register name	\$f22	\$f23	\$f24	\$f25	\$f26	\$f27
Standard alias/role	\$ft6	\$ft7	\$ft8	\$ft9	\$ft10	\$ft11
Substituted name	\$gf6/\$df2	\$gf7/\$sf2	\$gf8/\$df1	\$gf9/\$sf1	\$gf10/\$df0	\$gf11/\$sf0
Legacy register name	\$f28	\$f29	\$f30	\$f0	\$f1	\$f21
Standard alias/role	\$ft12	\$ft13	\$ft14	\$frv	\$frv2	\$fa5
Substituted name	\$lf0	\$lf1	\$lf2	\$lf3	\$lf4	\$lf5
Legacy register name	\$f20	\$f19	\$f18	\$f17	\$f16	\$f2
Standard alias/role	\$fa4	\$fa3	\$fa2	\$fa1	\$fa0	\$fs0
Substituted name	\$lf6	\$lf7	\$lf14	\$lf8	\$lf9	\$lf10
Legacy register name	\$f3	\$f4	\$f5	\$f6	\$f7	\$f8
Standard alias/role	\$fs1	\$fs2	\$fs3	\$fs4	\$fs5	\$fs6
Substituted name	\$lf11	\$lf12	\$lf13	\$lf15	\$lf16	\$lf17
Legacy register name	\$f9	\$f31				
Standard alias/role	\$fs7	\$f31				
Substituted name	\$lf18	\$f31				

Table H.2: Substitution table for FP register names.

H.7 Forward declarations, separate compilation and visibility

Another requirement was support for separate compilation. Provision for this in the C language exists through the notion of visibility, controlled via the keywords “extern” and “static” (or their absence), and forward declarations. Like thread functions and objects, we must allow the source code to control the visibility of thread programs and express forward declarations.

We enabled this in two steps:

1. we added an additional positional parameter to the “`sl_def`” construct where the word “static” can be expressed optionally, in which case it is injected accordingly in the expansion. The new form then becomes “`sl_def(FunName, Visibilityopt, Channels...opt)`”;
2. we added a new construct “`sl_decl`” which accepts the same positional parameters as “`sl_def`” and expands to a declaration (without a body).

Later on, we found out that we needed a different textual expansion for declarations of thread programs, and declarations of pointers to thread programs. We introduced an extra form “`sl_decl_fptr`” for this purpose, again with the same positional parameters as “`sl_def`.” Although there is no additional associated conceptual difficulty, we mention it here for completeness.

H.8 Extending bulk creation

At a language level, two features outside of the scope of this appendix are also supported with the “`sl_createsync`” construct: the specification of a *place identifier*, which is a named processing resource where the work should be executed (cf. chapter 11 and Appendix E), and extra *creation specifiers* for controlling non-function aspects of code generation (chapter 10). The final syntax interface settled upon was “`(, Placeopt, Startopt, Limitopt, Stepopt, Blockopt, Specifiersopt, FunName, Channels...opt)`” with an initial empty position reserved for custom compiler extensions.

As an attempt to more closely related to the original language proposal (Appendix G), we also split the “`sl_createsync`” construct into two syntax elements “`sl_create`” and “`sl_sync`,” bound into a single syntax rule as explained later in Appendix I.5.8.1. Between these two, a new form “`sl_seta`” symmetric to “`sl_geta`” and “`sl_setp`” presented above can provide source values to the channel endpoints. Otherwise, all the positional parameters to “`sl_createsync`” were retained in the definition of “`sl_create`.”

Of course, the syntactic substitution of the bulk creation operations for the “fused” interface, or as a C loop for the sequential version, must occur at the point “`sl_sync`” is expressed, not earlier. This is because the source values for channel endpoints may not fully determined prior to this point. This is not further discussed here, as there is no additional technical difficulty involved; for more details cf. the source code of the produced tools, or fig. 6.1 for a summary.

H.9 Support for function calls

Another layer of implementation efforts were dedicated to supporting regular function and procedure calls. These efforts occurred beyond the mandatory provision of private stacks to threads, discussed separately in chapter 9.

The first aspect was to determine how to compile separately a regular C function that can be called from different thread programs. The problem that needed to be addressed is that the register window layout is specific to each thread, and determined by the register configuration word (section 4.3.3.2), whereas the C function code must refer to the same register names in all use contexts. To allow a single code representation for a C function that could be invoked from two threads with different register layout, we had to:

1. constrain the layout of the visible window (section 4.3.3.4) so that the register names in the local synchronizer region would be the same across all threads; this is achieved by mapping the private register space always at the start of the ISA window, i.e. the logical names “\$I0...\$IN” must map to the machine names “\$0...\$N,” instead of $\$(G + S)...\$(G + S + N)$ ” as with e.g. the original G-D-S-L layout.
2. apply different post-processing stages to the assembly code of regular C functions than those applied to thread programs, to account for different patterns of register uses; this was achieved by sorting generated assembly codes prior to post-processing based on the presence of the “registers” directive.

Once function calls can be generated, a situation can occur where a function must save callee-save registers [JR81] before using them for computations. For this purpose the code generator would issue *spills* (stores to stack) in the function code prior to any other use of these registers. However if the function is called from a thread program, and the callee-save registers are not assigned a value in the thread program prior to the function call, it is possible that some of the registers are initially in the “empty” state (section 4.6.1.2), which would incur a deadlock when the spill is reached during execution.

To avoid this problem, we initially modified the post-processor to issue instructions to force all callee-save registers to a non-empty state at the start of a thread program. However, this resulted in a mandatory overhead to fill all the callee-save registers (26 instructions with both the Alpha and SPARC substrate ISAs), even when the registers may end up not being used at all (in most cases it is not possible to determine whether the registers are used or not, since the called function is usually compiled separately). Later on, we negotiated a change to the machine interface semantics instead, to mandate that all local registers are guaranteed upon thread start-up to be readable. This allowed us to avoid the overhead in software.

On a related note, saving and restoring callee-save registers is not necessary in the top-level thread function, since its private register window is dedicated; the post-processor was thus enhanced to remove these when they could be detected.

Finally, when targeting code for the SPARC substrate ISA, we had to address the conceptual conflict between SPARC’s sliding register windows [MAG⁺88] and our machine interface, introduced in section 4.5.4. The problem is that the SPARC machine interface guarantees that the **save** and **restore** instructions will always obtain a “fresh” set of local registers for a function, possibly at the expense of an overflow trap if the window pointer reaches an existing frame around in the register file. Since the implementations we had access to do not implement SPARC’s sliding windows and overflow traps, the **save** and **restore** instructions cannot be used at all in thread programs and all the function codes they call indirectly. Instead, we must translate all occurrences of “**save**” and “**restore**” in the assembly source using the substitutions in listings H.25 and H.26. As an optimization, we also automatically skip the saving and restoring instructions that target registers otherwise not used in the current function code.

```

1  save A, B, C
2  →
3  add A, B, %g1
4  std %i0, [%sp - 0]
5  std %i2, [%sp - 8]
6  std %i4, [%sp - 16]
7  std %fp, [%sp - 24]
8  mov %o0, %i0
9  mov %o1, %i1
10 mov %o2, %i2
11 mov %o3, %i3
12 mov %o4, %i4
13 mov %o5, %i5
14 mov %sp, %fp
15 mov %o7, %i7
16 mov %g1, C

```

Listing H.25: Substitution for SPARC's `save`.

```

1  restore A, B, C
2  →
3  add A, B, %g1
4  mov %i7, %o7
5  mov %fp, %sp
6  ldd [%sp - 24], %fp
7  mov %i5, %o5
8  mov %i4, %o4
9  ldd [%sp - 16], %i4
10 mov %i3, %o3
11 mov %i2, %o2
12 ldd [%sp - 8], %i2
13 mov %i1, %o1
14 mov %i0, %o0
15 ldd [%sp - 0], %i0
16 mov %g1, C

```

Listing H.26: Substitution for SPARC's `restore`.

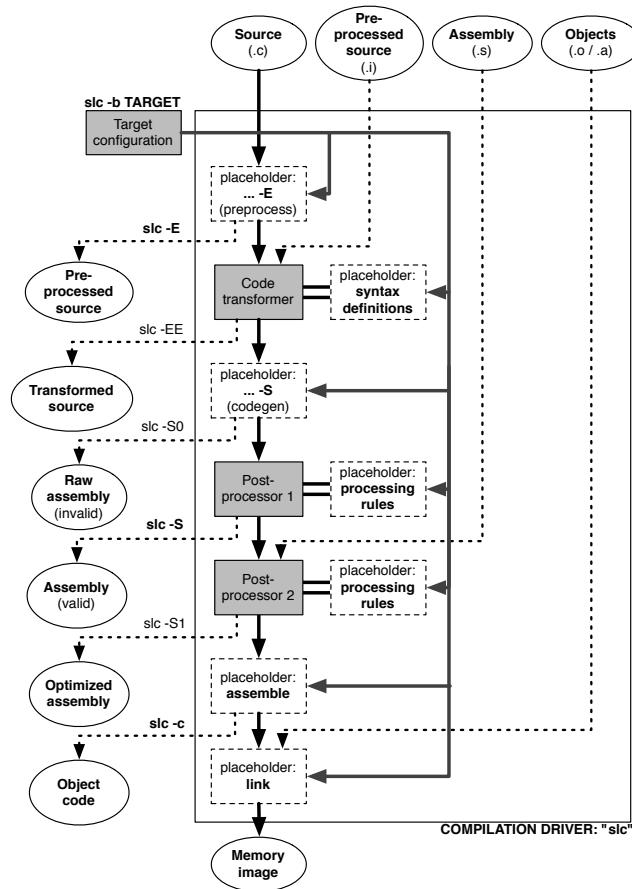


Figure H.3: The "slc" compilation pipeline and driver.

H.10 Resulting compilation chain

The resulting tool infrastructure is illustrated in fig. H.3.

Our translation pipeline contains the underlying C pre-processor, our code transformer in charge of performing context-free syntax substitutions on the C source, the underlying C code generator, two stages of post-processing on the assembly code, and the assembler and linker programs. While the previous sections describe only one post-processing stage implementing all transformations on the assembly source, we actually split up this into two stages, the latter containing all transforms that are useful to apply to hand-written assembly. These include the addition of the "swch" annotation, and the removal of family configuration instruction that use the default values in hardware.

This pipeline is further *parameterized* by the target architecture, using the driver command-line parameter "-b": the choice of the underlying C compiler and pre-processor, which syntax substitution and post-processing rules should be applied, and which assembler and linker programs to use can be configured. This way we can use the same tool interface to compile towards various implementations of the new architecture, e.g. to different ISAs, and a

Target alias (“-b”)	Description
mta	New architecture with the Alpha ISA, MGSim, providing “detached” creation.
mts	New architecture with the SPARC ISA, UTLEON3, providing “fused” creation.
mtsn	New architecture with the SPARC ISA, MGSim, providing “detached” creation.
seqc	Sequential execution on the host platform.
hrt	Execution with a software concurrency framework [Mat10], providing “detached” creation.
pt1, hls	Execution with two other frameworks [vTJLP09, UvTJ11]; these provide “detached” creation but with “hanging” synchronizer mappings, so they are treated as a “fused” creation interface for code generation.

Table H.3: Supported compilation targets at the time of publication of this book.

legacy “host” platform in pure sequential mode, where the post-processing stages are entirely disabled. A summary of the target configurations we ended up supporting is described in table H.3.

We also took care of recognizing most of the GNU CC command-line interface and external semantics. This allowed us to use the new “**s1c**” command as a drop-in replacement, *without changes to existing build systems*, when we ported third-party software programs and libraries. In particular our driver recognizes the common flags “-E” to effect pre-processing only, “-S” to stop after code generation (but ensuring the assembly source is valid for the target architecture), and “-c” to stop after producing object code. The extra flags “-EE,” “-S0” and “-S1” are provided for completeness and ease of troubleshooting the compilation chain itself.

Appendix I

SL Language specification

Abstract

This appendix describes the SL language introduced in chapter 6 in the style of [II99, II11b].

Contents

I.1	Prologue	318
I.2	Terms and definitions	318
I.3	Conformance	318
I.4	Environment	319
I.5	Language	320
I.6	Library	332

I.1 Prologue

This appendix contains references to terms, definitions and concepts defined in [II99, II11b], which should therefore be considered bound to this specification.

I.2 Terms and definitions

The following terms are defined in section 3 of [II99, II11b] and are reused in this specification: “argument,” “constraint,” “implementation,” “implementation limit,” “object,” “parameter,” “undefined behavior,” “unspecified behavior,” “value.”

The following terms are syntactic / semantic constructs defined in [II99, II11b], section 6, and are reused in this specification: *block item*, *compound statement*, *constant expression*, *conversion*, *declaration specifier*, *declaration*, *definition*, *expression*, *function designator*, *identifier*, *linkage*, *name space*, *processing environment*, *scope*, *side effect*, *statement*, *storage class specifier*, *storage duration*, *translation environment*, *type*, *type compatibility*, *type specifier*, *universal characters*, *visibility*.

As a clarification to [II99, II11b], we further detail the semantics of C with regards to objects, storage duration (lifetime), addressability and mutability in Appendix F.

We did not consider the concurrency features in [II11b] as our work predates this specification. Instead, with [II99] we define the following extra terms:

logical thread a sequential unit of work defined at run-time by a thread program entry point and a channel endpoint interface definition;

family the set of logical threads defined by the execution of a single create construct; this corresponds to the definition of “family” in section 7.2.2.2;

logical thread index integer value that uniquely identifies a thread within a family; the set of logical thread indexes is defined when execution reaches a create construct.

thread program a program suitable for execution by a logical thread;

thread function synonymous to “thread program”.

thread parameter an incoming “global” channel endpoint in a thread program, or a pair of incoming/outgoing “shared” channel endpoints.

thread argument an outgoing “global” channel endpoint, or a pair of outgoing/incoming “shared” channel endpoints, in a creating thread respective to the created family.

parameter or argument kind the kind of channel that the parameter or argument endpoint is connected to, either “shared” (daisy-chained between logical threads) or “global” (from creating thread to all created threads).

thread function prototype the specification of the number, kind and type of thread parameters in a thread function declaration or definition.

sibling endpoints for an incoming channel endpoint, the outgoing endpoint at the other side of the channel; for an outgoing channel endpoint, the incoming endpoints at the other side of the channel.

I.3 Conformance

- 1 Section 4 from [II99, II11b] applies.

Side note I.1: About the compatibility of our implementation.

Our implementation of SL is a *conforming freestanding implementation* as per [II99, 4§6]. Later in our work we added library features to increase its compatibility as a conforming freestanding implementation as per [III1b, 4§6] and a *conforming hosted implementation* as per [II99, II11b], but this support is yet incomplete. We discuss this further in section 6.4.3.

I.4 Environment

- 1 SL’s translation and processing environments are defined like C’s environment.
- 2 Additionally, M4 is run as an additional pre-processor after C pre-processing and before translation.
- 3 In the rest of this section, the corresponding clauses from [II99, II11b] apply, unless explicitly modified or extended.

I.4.1 Program execution

- 1 [II99, 5.1.2.3§1] and [III1b, 5.1.2.3§1] apply.
- 2 [II99, 5.1.2.3§2] and [III1b, 5.1.2.3§2] apply with “at certain specified points in the execution sequence called sequence points [...]” changed to “at certain specified points in the execution sequence of a thread program, called sequence points [...]” That is to say, we restrict this clause so that the visibility of side effects is restricted to the rest of the execution of the same thread program, not concurrent activities.
- 3 Also, creating or waiting for termination of a family, reading from or writing to inter-thread channel endpoints, or calling a function that does any of those operations are all side-effects.
- 4 [III1b, 5.1.2.3§3] applies, insofar as the proposed relations apply to evaluations performed by *logical* threads as per our definition in I.2.
- 5 [II99, 5.1.2.3§3] and [III1b, 5.1.2.3§4] apply.
- 6 We leave unspecified whether the concept of “signal” exists in SL, and thus whether and how [II99, 5.1.2.3§4] and [III1b, 5.1.2.3§5] apply.
- 7 [II99, 5.1.2.3§5–7] and [III1b, 5.1.2.3§6–8] apply.

I.4.1.1 Multi-threaded executions and data races

- 1 [III1b, 5.1.2.4§1–4] apply.
- 2 [III1b, 5.1.2.4§5–10] do not apply directly: we discuss support for C’s “atomic objects” and their synchronizing operations further in chapter 7.
- 3 [III1b, 5.1.2.4§11–28] apply, insofar as C’s atomic objects are not supported directly, and the `kill_dependency` macro is not supported in SL.
- 4 The abstract machine for SL does not use *memory*, and thus C’s “objects” as per [II99, II11b] and Appendix F, for *synchronization* between threads. Instead, synchronization between threads, and thus the “*inter-thread happens before*” relation between actions described in [III1b, 5.1.2.4§16], is negotiated by programmable devices (dataflow channel endpoints) that are physically independent from memory. This divergence from [III1b] entails that operations that write to an object are not necessarily visible to subsequent read operations, even if the write and read operations are otherwise synchronized. We discuss this further in chapter 7.

I.4.2 Translation limits

- 1 C's translation limits apply.
- 2 Also, the implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:
 - 127 “global” channel endpoint definition in a thread program declaration or definition, or family creation;
 - 1 “shared” channel endpoint pair definition in a thread program declaration or definition, or family creation;
 - 0 levels of syntactic nesting of “`sl_create`” and “`sl_sync`” constructs, that is programs where the program text between the two constructs does not contain further occurrences of the constructs.

I.5 Language

I.5.1 Notation

- 1 We will reuse the syntax notation from [II99, II11b].
- 2 In addition, we will use the symbol “-” to denote continuation of a syntax pattern on the following line.

I.5.2 Concepts

I.5.2.1 Scope of identifiers

- 1 *Identifiers* can denote what they denote in C ([II99, 6.2.1], [II11b, 6.2.1]);
- 2 Also, identifiers can denote thread parameters and arguments.
- 3 Identifiers have the same *visibility* and *scope* semantics in SL as they have in C.

I.5.2.2 Linkage of identifiers

- 1 *Linkage* is defined in SL as in C ([II99, 6.2.2], [II11b, 6.2.2]).
- 2 Identifiers that denote channel endpoints have no linkage.

I.5.2.3 Name space of identifiers

- 1 *Name spaces* are defined in SL as in C for all identifiers except those that denote channel endpoints ([II99, 6.2.3], [II11b, 6.2.3]).
- 2 Identifiers for channel endpoints have a distinct name space from C label names, tags, members and ordinary identifiers.
- 3 Also, identifiers that denote thread parameters have a different name space than identifiers that denote thread arguments.

For example in the following fragment:

```

1 sl_def(foo , void , sl_glparm(int , x) ) {
2     float x;
3     sl_create( , , , , , foo , sl_glarg(int , x) );
4     sl_sync( );
5     // <-- here
```

```

6 }
7 sl_endif

```

at the location specified by “here,” both the variable “x,” the thread parameter “x” and the thread argument “x” can be used (the latter two resp. with “sl_getp” and “sl_setp”, cf. hereafter).

I.5.2.4 Storage duration of objects

- 1 *Storage duration* is defined in SL as for C objects ([II99, 6.2.4], [III1b, 6.2.4]), insofar as [III1b, 6.2.4§4,6.7.1]’s “_Thread_local” storage-class specifier is not supported in SL.
(note that since channel endpoints are not objects, they do not have a storage duration)

I.5.2.5 Types

- 1 *Types* are defined in SL as in C ([II99, 6.2.5], [III1b, 6.2.5]); however the “_Atomic” qualifier from [III1b, 6.2.5§27] is not supported in SL as discussed in chapter 7.
- 2 Also, SL adds a new type category: *thread function types*, which are separate from C’s function types. A thread function type is characterized by its attributes and the number, *kind* (“shared” or “global”) and type of its thread parameters, collectively called its prototype.
- 3 A thread function type shall not be incomplete.
- 4 Thread function types can be used for type derivation like function types.
- 5 Any of C’s *scalar types* ([II99, 6.2.5§21], [III1b, 6.2.5§21],) can be used to define thread parameters and arguments¹.

We furthermore intendedly leave *unspecified* whether channels can be defined using any of C’s other types (arrays, aggregates, functions), pending further research in that direction.

I.5.2.6 Representation of types

- 1 The *representation of types* is defined in SL as in C ([II99, 6.2.6], [III1b, 6.2.6]).
(since channel endpoints themselves are not objects, they do not have a representation, although the values read from or written to the endpoints do have a representation)

I.5.2.7 Compatible types and composite types

- 1 *Compatible and composite types* are defined in SL as in C ([II99, 6.2.7], [III1b, 6.2.7]).
- 2 Additionally, compatibility and composition is extended to *thread function types*. However:
 - thread function types and C’s function types are not mutually compatible;
 - two thread function types are not compatible if their attributes differ, or if their argument kind and type differ in any manner when compared one to one in their order of appearance in the function’s prototype.

¹In C, a pointer-to-array type is a scalar type, as well as pointer-to-function.

I.5.2.8 Alignment of objects

- 1 [III1b, 6.2.8§1] applies; however the `_Alignas` keyword is not supported.
- 2 [III1b, 6.2.8§2–4] apply, insofar as `_Alignof` and `max_align_t` are not supported.
- 3 [III1b, 6.2.8§5–7] apply.

I.5.3 Conversions

- 1 *Conversions* are defined in SL as in C ([II99, 6.3], [III1b, 6.3]).
- 2 Additionally, automatic conversions are extended to *thread function types* as follows: unlike C function designators, a thread function designator with type “thread function” is never promoted automatically to an expression that has type “pointer to thread function.”

I.5.4 Lexical elements

- 1 *Lexical elements* are defined in SL as in C ([II99, 6.4], [III1b, 6.4]), insofar as the universal character names and string literals from [III1b, 6.4.3,6.4.5] are not supported.
- 2 Additionally:
 - all identifiers beginning with “`__sl_`” and “`_sl_`” are reserved and cannot be used in programs;
 - the following pre-processor macros have pervasive semantics and must be considered as keywords (i.e. they must not be redefined, and they can be assumed to have the same semantics everywhere they appear): `sl_def`, `sl_undef`, `sl_create`, `sl_sync`, `sl_shparm`, `sl_glparm`, `sl_shfparm`, `sl_glpfparm`, `sl_glarg`, `sl_glfarg`, `sl_shfarg`, `sl_glfarg`, `sl_seta`, `sl_geta`, `sl_setp`, `sl_getp`, `sl_index`, `sl_decl_fptr`, `sl_lbr`, `sl_rbr`, `sl_typedef_fptr`.
- 3 Any keyword in C++ [III1a] not listed above is reserved in SL and cannot be used in programs².

I.5.5 Expressions

- 1 *Expressions* are defined in SL as in C ([II99, 6.5], [III1b, 6.5]), insofar as atomic objects (discussed above), [III1b, 6.5.1.1]’s generic selection with “`_Generic`” and [III1b, 6.5.3]’s alignment operator “`_Alignof`” are not supported in SL.
- 2 Additionally, SL introduces *thread argument and parameter access expressions*.

I.5.5.1 Thread argument and parameter access

Syntax

primary-expression:

```

...
sl_geta ( identifier )
sl_getp ( identifier )

```

²We reserve C++ keywords to reserve ourselves the ability to translate SL code to a C++-based substrate language without running the risk that valid uses of identifiers in the source SL code become invalid uses of keywords in C++.

Constraints

- 1 The identifier used with `sl_geta` must be a visible thread argument name (cf. I.5.8.1).
- 2 The `sl_geta` expression shall not appear within the create block item list of the create construct where the thread argument name is defined.
- 3 The identifier used with `sl_getp` must be a thread parameter name in the enclosing thread function.

Semantics

- 4 Thread argument and parameter access expressions are converted during evaluation to the value read from the corresponding channel endpoint; they correspond to the r and \bar{q} operations from section 7.2.
- 5 The type of a thread argument or parameter access expression is the declared type of the channel.
- 6 Each use of `sl_getp` generates a side effect (as per [II99, 5.1.2.3§2] and [III1b, 5.1.2.3§2]).
- 7 Execution passes the first sequence point following an occurrence of `sl_getp` no earlier than execution passes the `sl_setp` or `sl_seta` statement in the thread where the sibling channel endpoint is defined.
- 8 If execution reaches an expression using `sl_getp` after it has passed a `sl_setp` statement using the same thread parameter identifier, the behavior of the program becomes undefined.

(note that the *identifiers* themselves that denote thread arguments and parameters are not valid expressions)

I.5.6 Constant expressions

- 1 *Constant expressions* are defined in SL as in C ([II99, 6.6], [III1b, 6.6]).

I.5.7 Declarations

- 1 Any C declarations containing the storage class specifier “`typedef`,” built upon any of C’s types (i.e. not thread function types), any C declarations that declare or define an object or function, and any C declarations that declare a structure, enum or union type are valid in SL and have the same semantics as in C, insofar as the following features from [III1b] are not supported (cf. chapter 7):
 - [III1b, 6.7.2.4,6.7.3]’s “`_Atomic`” type specifier and qualifier;
 - [III1b, 6.2.4§3,6.7.1]’s “`_Thread_local`” storage class specifier;
 - [III1b, 6.7.4]’s “`_Noreturn`” function specifier;
 - [III1b, 6.7.5]’s “`_Alignas`” alignment specifier.
- 2 It is left unspecified here whether an implementation of SL must support [II99, III1b]’s variable length arrays or variably modified types.

(in [II99] this support was mandatory, but it is now an optional feature according to [III1b]. Our implementation makes a “best effort” in this direction but support for this feature was not thoroughly tested.)
- 3 Additionally, SL adds new constructs for *logical thread index declarations*, *thread function declarations*, *declarations of pointers to thread functions*, and *type name definitions for pointer types to thread functions*.

We intendedly leave *unspecified* whether any other of C’s declarations exists in SL.

I.5.7.1 Logical thread index declarations

Syntax

thread-index-declaration:
 sl_index (*identifier*) ;
declaration:
 ...
 thread-index-declaration

Constraints

- 1 A thread index declaration shall only appear in a thread function body.

Semantics

- 2 A thread index declaration declare a constant object of unspecified integer type in the current scope with the given name, whose value during execution is the logical thread index of the current thread.

I.5.7.2 Thread function declarations

Syntax

thread-function-declaration:
 sl_decl (*identifier* , *thread-specifiers*_{opt} ↯
 [, *thread-parameter-list*]_{opt}) ;
thread-parameter-list:
 thread-parameter-declaration
 thread-parameter-declaration , *thread-parameter-list*
external-declaration:
 ...
 thread-function-declaration

(note that this syntax is not suitable for use in a *struct-declaration-list*)

Semantics

- 1 A thread function declaration declares a thread function with the specified name and prototype, with external linkage unless the attribute “**sl__static**” is specified (cf. I.5.7.5).

I.5.7.3 Declarations of pointers to thread functions

Syntax

thread-function-pointer-declaration:
 sl_decl_fptr (*identifier* , *thread-specifiers*_{opt} ↯
 [, *thread-parameter-list*]_{opt}) ;
declaration:
 ...
 thread-function-pointer-declaration

(note that this syntax is not suitable for use in a *struct-declaration-list*)

Semantics

- 1 A thread function pointer declaration declares a pointer to thread function with the specified identifier and prototype, with external linkage unless the attribute “`__static`” is specified (cf. I.5.7.5).

I.5.7.4 Type name definitions for pointer types to thread functions

Syntax

```

thread-function-pointer-typedef:
    __typedef_ptr ( identifier
        [ , thread-specifiersopt ¬
        [ , thread-parameter-list ]opt ]opt ) ; ¬
declaration:
    ...
    thread-function-pointer-typedef

```

(note that this syntax is suitable for use within function bodies)

Semantics

- 1 A thread function pointer typedef define a typedef name that denotes a pointer to thread function type with the specified name and prototype.

For example:

```

1 __decl(foo , , __glparam(int , x));
2 ...
3 {
4     __typedef_ptr(ptype , , __glparam(int , y));
5     ptype p = &foo;
6     struct { ptype q; } r;
7     r.q = p;
8 }

```

I.5.7.5 Thread attributes and specifiers

Syntax

- 1


```

thread-specifiers:
    thread-specifier-item
    ( thread-specifier-list )
thread-specifier-list:
    thread-specifier-item
    thread-specifier-list , thread-specifier-item
thread-specifier-item:
    thread-specifier

```

thread-attribute
thread-specifier:
`sl__static`
thread-attribute:
 (yet undefined)

Semantics

- 2 The declaration or definition of a thread function can specify *attributes* and *specifiers*. Attributes are part of the prototype, while specifiers are not.
- 3 The thread specifier `sl__static` plays the same role as C's storage qualifier `static` on external declarations.

I.5.7.6 Thread parameter list

Syntax

thread-parameter-list:
thread-parameter-declaration
thread-parameter-list , *thread-parameter-declaration*
thread-parameter-declaration:
`sl_glparm` (*declaration-specifiers* , *identifier*)
`sl_glfparm` (*declaration-specifiers* , *identifier*)
`sl_shparm` (*declaration-specifiers* , *identifier*)
`sl_shfparm` (*declaration-specifiers* , *identifier*)

Semantics

- 1 A thread parameter declaration specifies channel endpoints for the thread program. The forms `sl_glparm` and `sl_glfparm` specify an incoming “global” channel endpoint; the forms `sl_shparm` and `sl_shfparm` specify an incoming/outgoing pair of “shared” channel endpoints.
- 2 The declaration specifiers part of a thread parameter declaration shall not contain a storage class specifier, nor the type qualifier `volatile`.
- 3 The type designated by the declaration specifiers (either directly, or indirectly via the use of a typedef name) in `sl_glparm` and `sl_shparm` shall be an integer type ([II99, 6.2.5§17], [II11b, 6.2.5§17]).
 Note that this includes pointers; as per section 7.3, care must be taken to organize the consistency of the objects pointed to separately.
- 4 The type designated by the declaration specifiers in `sl_glfparm` and `sl_shfparm` shall be either `float` or `double`.

I.5.8 Statements and blocks

- 1 *Statements and compound statements* are defined in SL as in C ([II99, 6.8], [II11b, 6.8]).
- 2 Also, SL adds *create constructs* and *outgoing communication statements* to C blocks.

I.5.8.1 Family creation

Syntax

- 1 *create-construct*:
- ```

 sl_create (, create-parameters , create-specifiersopt , \neg
 assignment-expression \neg
 [, thread-argument-list]opt) ; \neg
 create-block-item-listopt \neg
 sl_sync () ;

```
- create-parameters*:
- ```

    assignment-expressionopt , range-parameters

```
- range-parameters*:
- ```

 assignment-expressionopt , assignment-expressionopt , \neg
 assignment-expressionopt , assignment-expressionopt

```
- create-specifiers*:
- ```

    create-specifier
    ( create-specifier-list )

```
- create-specifier-list*:
- ```

 create-specifier
 create-specifier-list , create-specifier

```
- create-specifier*:
- ```

    thread-attribute

```
- thread-argument-list*:
- ```

 thread-argument-definition
 thread-argument-list , thread-argument-definition

```
- thread-argument-definition*:
- ```

    sl_glarg ( declaration-specifiers , identifieropt  $\neg$ 
        [ , assignment-expression ]opt )
    sl_glfarg ( declaration-specifiers , identifieropt  $\neg$ 
        [ , assignment-expression ]opt )
    sl_sharg ( declaration-specifiers , identifieropt  $\neg$ 
        [ , assignment-expression ]opt )
    sl_shfarg ( declaration-specifiers , identifieropt  $\neg$ 
        [ , assignment-expression ]opt )

```
- create-block-item-list*:
- ```

 create-block-item
 create-block-item-list create-block-item

```
- create-block-item*:
- ```

    statement
    create-construct

```
- block-item*:
- ```

 ...
 create-construct

```

- 2 The syntax form is described as follows: the word “`sl_create`” followed by an opening parenthesis, followed by a comma, followed by five optional assignment expressions separated by (mandatory) commas, followed by an optional create specifier list, followed by a comma,

---

**Side note I.2:** About the syntax of “`sl_create`” and “`sl_sync`”.

---

The proposed syntax fuses the “`sl_create`” construct with the “`sl_sync`” construct in a syntactic unit. This formalizes the fundamental distinction between SL and the original language proposal in Appendix G.

The proposed construct also differs from C statements in that it can only appear in an enclosing block (e.g. a function body or compound statement), not as lone statements where these are allowed (e.g. as one branch of a `if` statement). This is because it plays both the role of a declaration and a statement, and declarations are not valid where lone statements are allowed. Also, intuitively, as declaration, the identifiers they define must be reusable beyond the construct in the enclosing block.

Also, the construct defines a new phrase structure, the *create block item list*: this differs from C’s *block item list* ([II99, 6.8.2], [III1b, 6.8.2]) in that it does not allow interspersed declarations, unless they are contained in a compound statement.

---



---

**Side note I.3:** About pointers to thread functions.

---

In C, an expression with type “pointer to function” can be used in a function call expression, and it is automatically converted to designate the function pointed to. In SL, the thread program expression in a create construct must designate the thread function directly. To use a pointer to thread function, the pointer must be explicitly dereferenced with the unary `*` operator.

---

followed by an assignment expression, followed by an optional list of comma-separated thread argument definitions, followed by a closing parenthesis, followed by a semicolon, followed by an optional create block item list, followed by the word “`sl_sync`,” followed by an opening parenthesis, followed by a closing parenthesis, followed by a semicolon.

- 3 The first five (optional) assignment expressions are called collectively *family configuration expressions* and individually the *place*, *start*, *limit*, *step*, *block* expressions of the construct. The sixth assignment expression is called the *thread program expression*.
- 4 In the thread argument list, each argument definition contains declaration specifiers, an optional identifier and an optional assignment expression, separated by commas. The optional identifier is called the *argument name* and the optional assignment expression is called the *argument initializer*.

### Constraints

- 5 Each of the family configuration expressions, when specified, shall have an arithmetic type.
- 6 The thread program expression must have a thread function type which is compatible with the thread argument list in the number, kind and type of channel definitions.
- 7 Each argument initializer shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding declaration specifier.
- 8 The argument names shall not be used in any other create construct in the same scope.
- 9 A `goto` or `switch` statement expressed outside a create construct shall not refer to a label expressed within its create block item list; nor shall a `goto` or `switch` statement expressed within the create block item list refer to a label outside of the create construct.
- 10 A `break` or `continue` statement shall only appear within the create block item list of a create construct if the entire enclosing loop also appears inside the construct.
- 11 The `return` statement shall not be expressed in a create block item list.

(we leave implementation-defined whether any other statements than thread argument assignments and thread parameter assignments with a single identifier as right operand are valid in a create block item list)

## Semantics

- 12 There is a sequence point after each family configuration expression, after the thread program expression, after each argument initializer, before execution passes the “`sl_create`” into the inner create block item list, before execution reaches “`sl_sync`”, and before execution passes “`sl_sync`.”
- 13 If any expression used in the create construct has side effects, they will be effected before their respective sequence point is passed during execution. This includes initialization values for thread parameters that are present in the thread argument list.
- 14 A create construct maps to a  $c$  and  $s$  actions in the formalism from section 7.2: both are guaranteed to occur at some point after execution passes “`sl_create`” into the inner create block item list, and before execution passes “`sl_sync`.”
- 15 As a property of the family composition contract, the create construct defines a set of executions of the thread program, collectively called a *family*; each execution in the family is guaranteed to start no earlier than when execution in the creating thread passes beyond the “`sl_create`” part and into the create block item list (or the “`sl_sync`” part if the block item list is not expressed); and execution in the creating thread is guaranteed to pass beyond the last sequence point in the construct no earlier than the termination of all executions in the family.
- 16 It is implementation-defined whether the  $c$  action in the abstract machine occurs *before* execution reaches “`sl_sync`.” In particular, the side effects in the create block item list may dominate the start of the family in the precedence order  $\sim$ ; in this case, if execution never reaches that point in the creating thread, the family may not execute at all.
- 17 Each execution is associated with a unique *logical thread index*, which can be observed via a `sl_index` declaration in the designated thread program (cf. I.5.7.1).
- 18 The set of all logical thread indexes  $\mathcal{S}$  used the family is defined by the triplet of integer values  $(A, B, C)$ , defined by the integer conversion ([II99, 6.3.1.1], [III1b, 6.3.1.1]) of the start, limit, step configuration expressions, as follows:

$$\mathcal{S} = \begin{cases} \{A + nC \mid n \in \mathbb{N} \wedge A \leq A + nC \leq B\} & \text{if } C > 0 \\ \{A + nC \mid n \in \mathbb{N} \wedge B \leq A + nC \leq A\} & \text{if } C < 0 \end{cases}$$

The behavior of the program is undefined if  $\mathcal{S}$  is not finite when execution reaches the create construct (i.e. when  $C = 0$ ), or if any of the values in  $\mathcal{S}$  does not fit in one of the base integer types. If  $\mathcal{S} = \emptyset$ , no execution of the thread program takes place (in particular when  $A = B$ ).

- 19  $\mathcal{S}$  is ordered with the following total order:

$$\forall x \in \mathcal{S}, \forall y \in \mathcal{S} \quad x \angle y \Leftrightarrow \begin{cases} x < y & \text{if } C > 0 \\ x > y & \text{if } C < 0 \end{cases}$$

- 20 The logical index value  $x \in \mathcal{S}$  such that  $\forall y \in \mathcal{S} \setminus \{x\} \quad x \angle y$ , if it exists, is called the *first index* of the family.
- 21 The logical index value  $x \in \mathcal{S}$  such that  $\forall y \in \mathcal{S} \setminus \{x\} \quad y \angle x$ , if it exists, is called the *last index* of the family.

(the first and last indexes may be identical, e.g. when  $A = 0, B = 1, C = 1$ )

- 22 For any logical index value  $x \in \mathcal{S}$ , its *successor value* is the logical index value  $y \in \mathcal{S} \setminus \{x\}$  such that  $\forall z \in \mathcal{S} \setminus \{x, y\}, z \angle x \vee y \angle z$ . The last index has no successor.
- 23 The order  $\angle$  described here corresponds to the order  $\angle$  from section 7.2.2.2: the successor index value of a thread is the index of the successor thread in the family.

---

**Side note I.4:** Defining the index sequence in the abstract semantics.
 

---

The phrasing in clause I.5.8.1§18 purposefully avoids defining *how* the index sequence is computed, in order to prevent assumptions about the behavior of the machine in case of overflow. This way, we are able to abstract the index sequence away from the operational semantics of a particular implementation.

However it is compatible with the phrasing in section 4.3.2.3.

---

- 24 The executions in the family and the execution of the creating thread are related via channels, with a topology determined by the kind of each channel declaration:
- the outgoing endpoint of a “global” channel in the creating thread is connected to the incoming endpoint of the corresponding “global” channel in each execution of the designated thread program;
  - the outgoing endpoint of a “shared” channel in the creating thread is connected to the incoming endpoint of the corresponding “shared” channel pair in the execution associated with the first index of the family;
  - the outgoing endpoint of a “shared” channel pair in an execution associated with any index but the last is connected to the incoming endpoint of the corresponding “shared” channel pair in the execution associated with the successor index;
  - the outgoing endpoint of a “shared” channel pair in an execution associated with the last logical index is connected to the incoming endpoint of the corresponding “shared” channel in the creating thread.

An illustration of this topology is given in fig. I.1 and fig. I.2; control dependencies are given in dotted lines, “global” channels are indicated in blue and “shared” channels in orange. Although the schedule is unspecified, and the family thread executions can occur either simultaneously, in sequence, or out of index order (e.g. in fig. I.2), the channel topology and the relationship between “`sl_set`” and “`sl_get`” is guaranteed by the language semantics.

- 25 The run-time value  $b$  of the “block” family configuration expression, if it is non-zero, *constrains* the definition of the sequential segments of the family (section 7.2.2.2) by the execution environment, by requiring that no more than  $b$  separate sequential segments are defined for the family, per processor. This corresponds to the constrain mechanism introduced in section 4.3.2.2. This implies that no more than  $b \times P$  units of fair scheduling as used to run the threads in the family, where  $P$  is the actual number of processors involved in the creation at run-time. If  $b = 0$ , no constraint is expressed.
- 26 The “place” family configuration expressions does not influence the functional behavior of the program; it is described further in chapter 11.

### I.5.8.2 Outgoing communication statements

#### Syntax

```

thread-argument-assignment:
 sl_seta (identifier , assignment-expression) ;
thread-parameter-assignment:
 sl_setp (identifier , assignment-expression) ;
statement:
 ...
 thread-argument-assignment
 thread-parameter-assignment

```

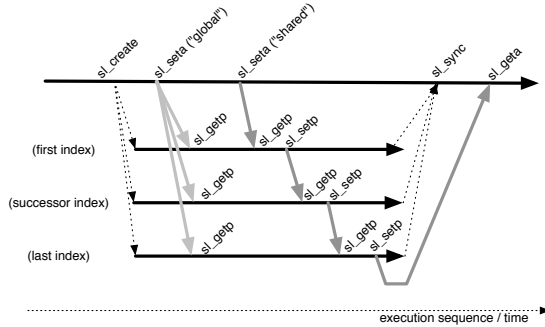


Figure I.1: Channel topology in a parallel thread family.

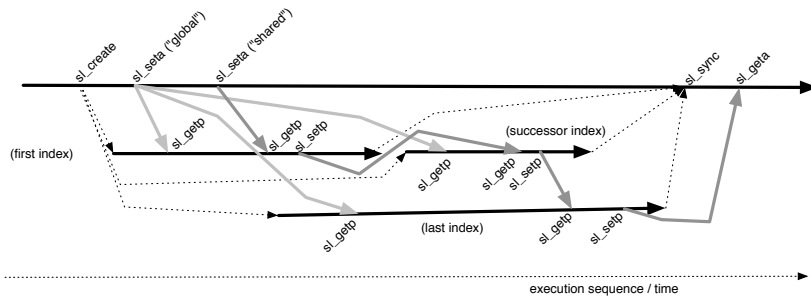


Figure I.2: Channel topology in a thread family with irregular schedule.

### Constraints

- 1 The identifier used with `sl_seta` must be a visible thread argument name (cf. I.5.8.1).
- 2 The `sl_seta` statement shall not appear outside of the create block item list of the create construct where the corresponding thread argument name is defined.
- 3 The identifier used with `sl_setp` must be a thread parameter name in the enclosing thread function.
- 4 The `sl_setp` statement shall not appear outside of a thread function body.
- 5 The right assignment expression in either `sl_setp` or `sl_seta` must be a suitable expression for use as the right operand of a virtual simple C assignment expression where the left operand would be an lvalue with the channel type ([I99, 6.5.16.1], [II11b, 6.5.16.1]).

### Semantics

- 6 When they are reached during execution, the thread argument and parameter assignment statements cause the output of the value of the specified expression to the channel end-point designated by the specified identifier; they correspond to the  $w$  and  $\bar{w}$  operations in section 7.2.
- 7 There is a sequence point before execution passes a thread argument or parameter assignment statement.
- 8 If execution reaches a thread argument or parameter assignment statement after it has passed another such statement designating the same channel endpoint, the behavior of the program becomes undefined.

### I.5.9 External definitions

- 1 Any of C's external declarations that declare or define external objects, any of C's external function declarations and definitions, any of C's external declarations with the storage class specifier `typedef`, and any of C's external declarations that declare a structure, enum or union type ([II99, 6.9], [III11b, 6.9]) are valid in SL with the same semantics as in C, with the same restrictions as in clause I.5.7§1 above.
- 2 Additionally, SL defines a new construct for *thread function definitions*.

#### I.5.9.1 Thread function definitions

##### Syntax

*thread-function-definition*:

```

sl_def (identifier [, attributesopt ¬
 [, thread-parameter-list]opt]opt) ¬
 compound-statement ¬
 sl_enddef

```

*external-declaration*:

```

...
thread-function-definition

```

##### Constraints

- 1 The identifier in thread functions definitions is in the same name space as C object and function names. Therefore, [II99, 6.9§3] and [III11b, 6.9§3] apply.
- 2 The constraints in I.5.7.6 apply.

##### Semantics

- 3 A thread function definition specifies the name of the thread function being defined, the identifiers of its channel endpoints, and the body of the thread program. The clauses in I.5.7.6 apply.

### I.5.10 Preprocessing directives

- 1 *Preprocessing directives* are defined in SL as in C ([II99, 6.10], [III11b, 6.10]).
- 2 In particular, an SL implementation may define any of [III11b, 6.10.8.3]'s conditional feature macros (e.g. `__STDC_NO_ATOMICS__`) with the same semantics.
- 3 Additionally, M4 [KR77] is run to filter the pre-processed text after pre-processing completes and before translation starts, with the M4 quotes changed to “[[” and “]]” to avoid conflicting with other uses of punctuation valid in C, and all predefined M4 macros renamed with the “m4\_” prefix to avoid conflicting with existing C identifiers.

## I.6 Library

All of C's reserved identifiers are reserved in SL ([II99, 7.1.3], [III11b, 7.1.3]).

Other library services available in SL are described in section 6.4.3.

# Appendix J

## QuickSort example

To exercise the dynamic mapping scheme described in chapter 10, we experimented with two implementations of the QuickSort algorithm with a cycle-accurate emulator of the target architecture.

### J.1 Benchmark program

Our two implementations are given in listing J.1 and listing J.2. The first implementation is the “naive” algorithm which spawns two separate families for each branch of the recursion; the second implementation uses a single family of two threads at each recursion step. The procedure implementing each recursion step is augmented to emit the array index of the values being examined: the position of the chosen pivot, and the positions of the successive values being swapped around the pivot.

### J.2 Input and baseline

We used three different input arrays, each consisting of 300 pseudo-random integers.

To establish a baseline, we also hand-coded a pure sequential version of the first algorithm which does not use concurrency in any way. We then ran this sequential version on the cycle-accurate machine emulation of the target architecture (MGSim).

Our results are illustrated in fig. J.1. These diagram represent memory activity over time. The horizontal axis represents time; the vertical axis represents cells in the array being sorted. A dot in the diagram indicates a memory read or write operation. The average time to result is between 40 and 50 kcycles.

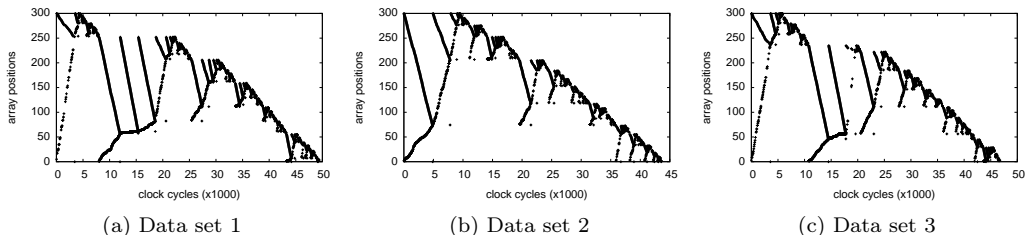


Figure J.1: Baseline: purely sequential algorithm as one thread.

---

```

1 #include <svp/testoutput.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <fcntl.h>
6 #define MARK_P1(x) output_uint((x<<2)|1, 0)
7 #define MARK_P2(x) output_uint((x<<2)|2, 0)
8 #define MARK_P3(x) output_uint((x<<2)|3, 0)
9
10 void swap(int *a, int *b) {
11 int t=*a; *a=*b; *b=t;
12 }
13
14 sl_def(sort, sl_static,
15 sl_glparm(int*, arr),
16 sl_glparm(size_t, beg), sl_glparm(size_t, end)) {
17 size_t beg = sl_getp(beg);
18 size_t end = sl_getp(end);
19 if (end > beg + 1) {
20 int *arr = sl_getp(arr);
21
22 /* compute pivot as median of
23 * start, end and middle values */
24 int x = arr[beg], y = arr[end-1],
25 z = arr[beg+(end-beg)/2];
26 int piv = ((x < y) && (y < z)) ? y
27 : (((y < x) && (x < z)) ? x : z);
28 int l = beg + 1, r = end;
29
30 while (l < r) {
31 /* organize values around pivot */
32 MARK_P1(1);
33 if (arr[l] <= piv)
34 l++;
35 else {
36 MARK_P2(1); MARK_P3(r-1);
37 swap(&arr[l], &arr[--r]);
38 }
39 }
40 MARK_P2(1-1); MARK_P3(beg);
41 swap(&arr[--l], &arr[beg]);
42
43 /* recurse */
44 sl_create(,,,,,sort, sl_glarg(int*,, arr),
45 sl_glarg(size_t,, beg), sl_glarg(size_t,, 1));
46 sl_create(,,,,,sort, sl_glarg(int*,, arr),
47 sl_glarg(size_t,, r), sl_glarg(size_t,, end));
48 sl_sync();
49 sl_sync();
50 }
51 }
52 sl_enddef
53
54 int array[100000];
55
56 sl_def(t_main, void) {
57 /* read N values from the file "data",
58 * N set via environment variable. */
59 size_t n = atoi(getenv("N"));
60 int fd = open("data", O_RDONLY);
61 for (ssize_t p = 0; p < n; ++p)
62 p += read(fd, array+p, (n-p)*sizeof(int));
63 close(fd);
64
65 /* perform the computation */
66 sl_create(,,,,,sort, sl_glarg(int*,, array),
67 sl_glarg(size_t,, 0), sl_glarg(size_t,, n));
68 sl_sync();
69 }
70 sl_enddef

```

---

Listing J.1: QuickSort benchmark in SL, classic algorithm.

---

```

1 #include <svp/testoutput.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <fcntl.h>
6 #define MARK_P1(x) output_uint((x<<2)|1, 0)
7 #define MARK_P2(x) output_uint((x<<2)|2, 0)
8 #define MARK_P3(x) output_uint((x<<2)|3, 0)
9
10 void swap(int *a, int *b) {
11 int t=*a; *a=*b; *b=t;
12 }
13
14 sl_def(sort2, sl__static, sl_glparm(int*,arr),
15 sl_glparm(size_t, beg1), sl_glparm(size_t, end1),
16 sl_glparm(size_t, beg2), sl_glparm(size_t, end2)) {
17
18 sl_index(i);
19 size_t beg = i ? sl_getp(beg2) : sl_getp(beg1);
20 size_t end = i ? sl_getp(end2) : sl_getp(end1);
21 if (end > beg + 1)
22 {
23 int *arr = sl_getp(arr);
24 /* compute pivot as median of
25 * start, end and middle values */
26 int x = arr[beg], y = arr[end-1],
27 z = arr[beg+(end-beg)/2];
28 int piv = ((x < y) && (y < z)) ? y
29 : (((y < x) && (x < z)) ? x : z);
30 int l = beg + 1, r = end;
31 /* organize values around pivot */
32 while (l < r) {
33 MARK_P1(1);
34 if (arr[l] <= piv)
35 l++;
36 else {
37 MARK_P2(1); MARK_P3(r-1);
38 swap(&arr[l], &arr[--r]);
39 }
40 }
41 MARK_P2(1-1); MARK_P3(beg);
42 swap(&arr[--l], &arr[beg]);
43 /* recurse */
44 sl_create(, , , 2, , , sort2, sl_glarg(int*, , arr),
45 sl_glarg(size_t, , beg), sl_glarg(size_t, , 1),
46 sl_glarg(size_t, , r), sl_glarg(size_t, , end));
47 sl_sync();
48 }
49 }
50 sl_enddef
51
52 int array[100000];
53
54 sl_def(t_main, void) {
55 /* read N values from the file "data",
56 * N set via environment variable. */
57 size_t n = atoi(getenv("N"));
58 int fd = open("data", O_RDONLY);
59 for (ssize_t p = 0; p < n; ++p)
60 p += read(fd, array+p, (n-p)*sizeof(int));
61 close(fd);
62
63 /* perform the computation */
64 sl_create(, , , , sort2, sl_glarg(int*, , array),
65 sl_glarg(size_t, , 0), sl_glarg(size_t, , n),
66 sl_glarg(size_t, , 0), sl_glarg(size_t, , 0));
67 sl_sync();
68 }
69 sl_enddef

```

---

Listing J.2: QuickSort benchmark in SL, families of two threads.

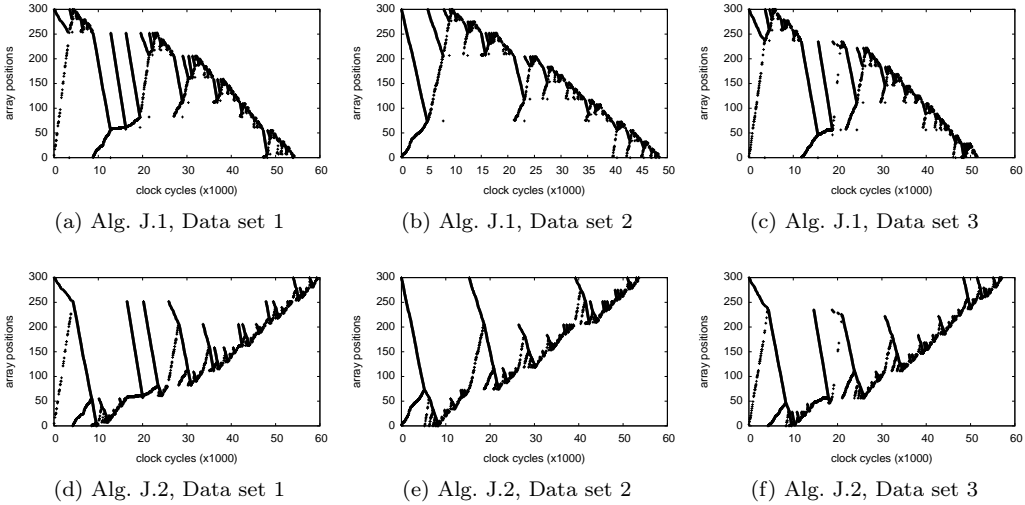


Figure J.2: Execution on 1 core with 1 family context.

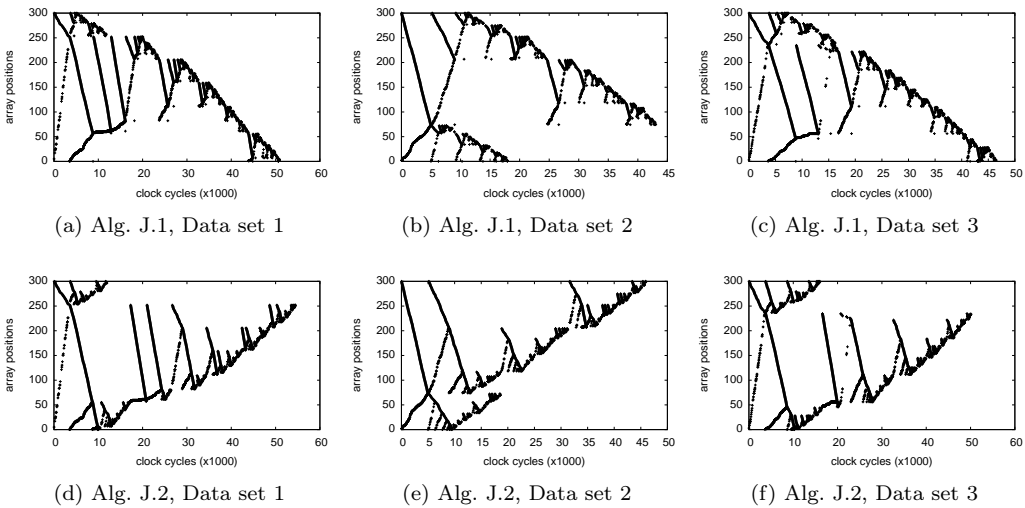


Figure J.3: Execution on 1 core with 3 family contexts.

### J.3 Multi-threaded behavior

We then executed our concurrent version, with only 1 family context available in the underlying architecture. The results are shown in fig. J.2. The overhead of the conditional on the availability of concurrency resources increases the time to result to between 50 and 60 kcycles.

We then executed the same programs over a system configured to offer only 3 family contexts to programs, on one core. The results are shown in fig. J.3. Despite the overhead,

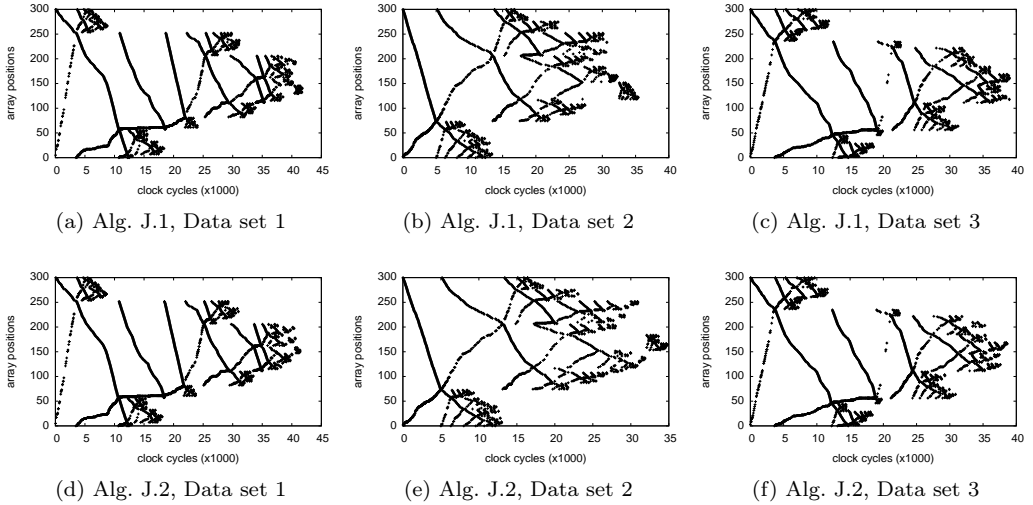


Figure J.4: Execution on 1 core with 31 family contexts.

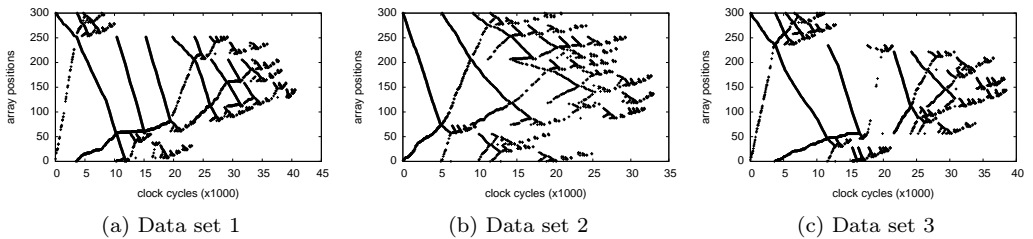


Figure J.5: Execution on 1 core with 31 family contexts, using Algorithm J.1 and a threshold on concurrency creation.

the overlap of multiple threads in the pipeline reduces bubbles and the total execution time becomes equal or below the overhead.

With 31 family contexts, which is the standard configuration of the target implementation, we obtain the results in fig. J.4 for one core: the time to result is reduced to between 35 and 45 kcycles, i.e. an improvement of 10% to 15% over the baseline.

We also tried to compensate the overhead of testing on resource availability by only using the concurrency construct if the amount of work (size of the sub-array) is larger than 16 items. Our results with this attempt are shown in fig. J.5: although there are less failed allocations overall, the overhead of the additional conditionals (6 extra instructions) is nearly equal to the cost of a failed allocation, resulting in a negligible gain overall.



## Appendix K

# Mandelbrot set approximation

This appendix provides the source code for the example heterogeneous workload from section 13.3.

---

```
1 sl_def(mandel, ,
2 sl_glfparm(double, four),
3 sl_glfparm(double, xstart), sl_glfparm(double, ystart),
4 sl_glfparm(double, xstep), sl_glfparm(double, ystep),
5 sl_glfparm(uint16_t, xres), sl_glfparm(size_t, icount)) {
6 // compute the complex point from the
7 // logical thread index
8 sl_index(i);
9 uint16_t xb = i % sl_getp(xres), yb = i / sl_getp(xres);
10 double cx = sl_getp(xstart) + xb * sl_getp(xstep);
11 double cy = sl_getp(ystart) + yb * sl_getp(ystep);
12
13 double zx = cx, zy = cy;
14 size_t v;
15 for (v = 0; v < sl_getp(icount); ++v) {
16 // iterate $z := z^2 + c$
17 double q1 = zx * zx, q2 = zy * zy;
18 if ((q1 + q2) >= sl_getp(four))
19 break;
20 double t = q1 - q2 + cx, q3 = zx * zy;
21 zx = t; zy = 2 * q3 + cy;
22 }
23 // pseudo-use of 'v' to prevent the compiler from erasing the loop as dead code.
24 asm volatile (" : :r"(v));
25 } sl_enddef
```

---

Listing K.1: Computation kernel executed by each logical thread.

---

```
1 sl_def(work) {
2 sl_create(, , /* logical index range: */ 0, NTHREADS_TOTAL, 1,
3 /* max nr. of threads / core: */ THREADS_PER_CORE, ,
4 mandel,
5 sl_glfarg(double, , 4.0),
6 sl_glfarg(double, , X_START), sl_glfarg(double, , Y_START),
7 sl_glfarg(double, , X_STEP), sl_glfarg(double, , Y_STEP),
8 sl_glfarg(uint16_t, , X_NPOINTS), sl_glfarg(size_t, , MAXITER));
9 sl_sync();
10 } sl_enddef
```

---

Listing K.2: Workload implementation using an even distribution.

---

```

1 sl_def(mandelouter, ,
2 sl_glfparm(double, xstart),
3 sl_glfparm(double, ystart),
4 sl_glfparm(double, xstep),
5 sl_glfparm(double, ystep),
6 sl_glparm(size_t, npoints),
7 sl_glparm(size_t, blocksize),
8 sl_glparm(uint16_t, xres),
9 sl_glparm(size_t, icount))
10 {
11 sl_index(p);
12 size_t n_cores = sl_placement_size(sl_default_placement());
13 sl_create(,
14 // execute on the local core:
15 PLACE_LOCAL,
16 // logical index range:
17 p, sl_getp(npoints)+p, n_cores,
18 // number of threads per core:
19 sl_getp(blocksize), ,
20 mandel,
21 sl_glfarg(double, , 4.0),
22 sl_glfarg(double, , sl_getp(xstart)),
23 sl_glfarg(double, , sl_getp(ystart)),
24 sl_glfarg(double, , sl_getp(xstep)),
25 sl_glfarg(double, , sl_getp(ystep)),
26 sl_glarg(uint16_t, , sl_getp(xres)),
27 sl_glarg(size_t, , sl_getp(icount)));
28 sl_sync();
29 }
30 sl_endif
31
32 sl_def(work)
33 {
34 size_t n_cores = sl_placement_size(sl_default_placement());
35 sl_create(, ,
36 // logical index range:
37 0, n_cores, 1,
38 // one thread per core:
39 1, ,
40 mandelouter,
41 sl_glfarg(double, , X_START),
42 sl_glfarg(double, , Y_START),
43 sl_glfarg(double, , X_STEP),
44 sl_glfarg(double, , Y_STEP),
45 sl_glarg(size_t, , NTHREADS_TOTAL),
46 sl_glarg(size_t, , THREADS_PER_CORE),
47 sl_glarg(uint16_t, , X_NPOINTS),
48 sl_glarg(size_t, , MAXITER));
49 sl_sync();
50 }
51 sl_endif

```

---

Listing K.3: Workload implementation using a round-robin distribution.