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

Citation for published version (APA):
Chapter 13

Core evaluation
—Lessons learned about hardware microthreading

Abstract

Once a platform is defined around an architectural innovation, effort must be invested into engaging with the audience and gaining feedback about the innovation. In this chapter, we provide an example of this interaction. In our ecosystem, a comprehensive evaluation of the architecture and its implementations was realized. We highlight some key results from this evaluation, for two purposes besides the evaluation results themselves. One purpose is to document how the platform definition from part II is practically related to the actual evaluation work, i.e. what interactions actually took place. The other is to illustrate that the interaction with our audience has enabled early feedback on the architecture design, as suggested in section 5.3.

Contents

13.1 Introduction .............................................................. 204
13.2 Evaluation efforts from the ecosystem .............................. 204
13.3 Separate impact of multi-threading and load balancing .......... 210
13.4 Applicability to irregular functional concurrency ............... 213
13.5 Optimization of performance per watt .......................... 215
13.6 Applicability to throughput-oriented applications ............... 217
13.7 Issues of system bandwidths, design trade-offs ................ 218
13.8 Relevance of thread-to-thread sharings .......................... 220
Summary ................................................................. 221
13.1 Introduction

As we explained in section 5.4.2, we worked in an ecosystem specifically set up to evaluate the innovation from part I. We faced three direct audiences: a research partner organization in charge of demonstrating the benefits of SAC [GS06] to program the proposed architecture, a partner organization in charge of discovering fine-grained concurrency in plain C loops, and various individuals in charge of hand-writing benchmarks using C and our proposed language extensions. In section 13.2 below, we explain through a running example the methodology used by our audience to carry out the platform evaluation. We also explain how our contributions from part II were useful in this context.

While we participated in these activities merely as support staff, we gained insight into the architecture. As this insight has not yet been published elsewhere, we share it in the remainder of this chapter. In particular:

- in section 13.3, we explain how we can separate the impact of multithreading from the impact of multi-core execution to understand performance results;
- in section 13.4, we identify dynamically heterogeneous, functional concurrency and introduce an architectural feature that we co-designed to optimize this use case;
- in section 13.5, we illustrate issues of power usage;
- in section 13.6, we illustrate throughput-oriented workloads;
- in section 13.7, we identify high-level issues shared by other multi-core designs;
- in section 13.8, we identify that a hallmark feature of the proposed design, the sharing of synchronizers by adjacent thread contexts, is actually of limited use and could be advantageously removed in favor of other, simpler features.

13.2 Evaluation efforts from the ecosystem

The benchmarking activities were organized around a single common theme: produce speedup diagrams that demonstrate the scalability of a single code representation across different hardware configurations. To achieve this, all the benchmarks have been written to share a common pattern:

1. load input data in memory;
2. sample performance counters;
3. execute a workload;
4. sample performance counters, and report the differences;
5. optionally, execute steps 2 to 4 multiple times.

Each program is then:

1. compiled once using our tools from chapters 6 and 8,
2. run multiple times on our platform from chapter 5, by placing it at start-up on core clusters of different sizes (from 1 to 64 cores) using the primitives from chapter 11.

Most programs further required the features detailed in chapters 9 and 10. The outcome for each benchmark is a series of samples which report the time to completion, the number of instructions executed, and other variables relevant to the evaluation of a processor architecture.

In addition to the features from part II, we implemented the following additional operating software for the benchmarking activities:
13.2. EVALUATION EFFORTS FROM THE ECOSYSTEM

Side note 13.1: About the relevance of the Livermore loops.

We acknowledge that the Livermore kernels are extremely small kernels unrepresentative of contemporary large applications. They were designed to be representative of loops in large high-performance applications and were selected/designed to test how effective vectorising compilers were at recognising concurrency in these applications. This benchmark suite is nowadays mostly superseded by newer developments that also test workloads from outside the HPC community, e.g. the more diversified Standard Performance Evaluation Corporation (SPEC) and NASA Advanced Supercomputing (NAS) benchmark suites. We discuss this further in chapter 15.

```fortran
COMMON /SPACE1/ U(1001), X(1001), Y(1001), Z(1001)
COMMON /SPACE2/ Q, R, T
...
cdir$ ivdep
1007 DO 7 k= 1,n
6    X(k)= U(k ) + R*( Z(k ) + R*Y(k )) +
7      1 T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
8      2 T*( U(k+6) + Q*( U(k+5) + Q*U(k+4))))
9    7 CONTINUE
```

Listing 13.1: FORTRAN code for the Livermore loop 7.

- a resource management service that reserves a cluster of cores upon system start-up and starts the benchmark program’s main function on this cluster;
- a performance counter sampling framework providing a uniform API to programs across all target implementations;
- a custom data input API able to load large arrays of data from files.

Beyond measuring performance over multiple core cluster sizes, each benchmark was also used to experiment with different compiler optimization flags, both at the higher-level SAC or parallelizing C compiler, and the SL tool chain. Each benchmark was also used to experiment with different architecture parameters, e.g. number of cores, cache sizes, etc. These benchmarking activities were spread over the ecosystem; an exhaustive report of all the benchmark results would be outside of our scope. Instead, we focus below on one benchmark to illustrate how the work was carried out in our technical framework.

13.2.1 Running example: Livermore loops

The Livermore FORTRAN kernels [McM86] are a sequence of 24 algorithms taken from scientific code. Their reference implementation is a FORTRAN program that exercises all 24 algorithms multiple times and computes a statistical report of their performance.

13.2.2 Implementation

Each of the 24 kernels was extracted individually from the FORTRAN implementation. One partner rewrote the kernels using SAC; separately, using a translation of the FORTRAN code to C as a basis, one partner used their parallelizing C compiler to automatically discover concurrency in the sequential C code and replace loops by uses of our language extensions, whereas we did the same work manually. We depict this implementation work in fig. 13.1.
Figure 13.1: Implementing the Livermore loop benchmarks using our proposed framework.

```
1 double u[1001], x[1001], y[1001], z[1001];
2 double q, r, t;
3 ...  
4 for ( k=0 ; k<n ; k++ ) {
5     x[k] = u[k] + r*( z[k] + r*y[k] ) +
6         t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
7             t*( u[k+6] + q*( u[k+5] + q*u[k+4] ) ) );
8 }
```

Listing 13.2: Sequential C code for the Livermore loop 7.

```
1 double[+] Loop7( int n, double q, double r,
2             double t, double[1001] u, double[1001] y,
3             double[1001] z);
4 double[+] Loop7( int n, double q, double r,
5             double t, double[+] u, double[+] y,
6             double[+] z)
7 {
8     a = u + r * ( z + r * y)
9         + t*(shift([-3], u) + r*(shift([-2], u) + r*shift([-1], u))
10            + t*(shift([-6], u) + q*(shift([-5], u) + q*shift([-4], u))));
11    return (take([n], a), inter);
12 }
```

Listing 13.3: SAC code for the Livermore loop 7.
13.2. EVALUATION EFFORTS FROM THE ECOSYSTEM

To illustrate further, we focus on one particular kernel, the equation of state fragment, whose original FORTRAN code is given in listing 13.1. This was separately translated to an equivalent C loop (listing 13.2) and parallel SAC code (listing 13.3); we then manually encapsulated the C loop body in a thread program and to produce the concurrent SL version in listing 13.4. Compared to the C code, we explicitly lift the reference to the globally declared variables as thread program channels in our SL code to avoid an external symbol reference in every thread, because our implementation strategy prevents the underlying C compiler from automatically detecting common sub-expressions across thread functions. Otherwise, no difficulty is introduced: the sequential loop of \( n \) iterations is replaced by a family creation of \( n \) logical threads. Another Livermore loop example using semi-explicit work placement to perform parallel reductions was also provided in chapter 11.

13.2.3 Results

Some example results for this benchmark are illustrated in figs. 13.2 to 13.4. Both the cycle-accurate, many-core, microthreaded platform emulation and a legacy architecture were used for comparison. The system characteristics of the legacy platform and the microthreaded platform for the results of this chapter are listed in table 13.1.

The code was compiled once (fig. 13.1) using version 3.6b of the SL tool chain, relying on GCC version 4.5 as an underlying code generator for the microthreaded architecture and GCC 4.2.1 to compile the sequential C code to the legacy architecture. The default compiler settings were used for optimization (“-O2”).

![](image)
For this benchmark, the input $n = 990$ was used as per the original Livermore benchmark specification. We used two baselines for comparison: the first is the behavior of the sequential C code on the legacy system, and the second is a hand-written, hand-tuned raw assembly program for the microthreaded platform written by an architecture expert.

There are different types of observations to draw from these results, depending on the audience.

### 13.2.3.1 Observations from the architect’s perspective

The results have illustrated that this specific implementation is able to scale performance for small workloads, e.g. 990 microthreads in the results above (each performing one iteration of the original loop) over multiple cores, up to dozens of cores, e.g. 32 cores above. This is remarkable as such small workloads (less than $10^5$ instructions) would not be able to compensate concurrency management overheads ($10^6$ instructions or more) on a legacy
13.2. EVALUATION EFFORTS FROM THE ECOSYSTEM

Figure 13.4: Floating-point performance for the Livermore loop 7.
Problem size and baseline are described in section 13.2.3.

<table>
<thead>
<tr>
<th>Legacy platform</th>
<th>Microthreaded platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>MacBookPro7,1</td>
</tr>
<tr>
<td>Processor chip</td>
<td>Intel Core 2 Duo P8600</td>
</tr>
<tr>
<td>Core</td>
<td>Intel Penryn</td>
</tr>
<tr>
<td>micro-architecture</td>
<td></td>
</tr>
<tr>
<td>Issue width</td>
<td>4</td>
</tr>
<tr>
<td>ISA</td>
<td>x86-64</td>
</tr>
<tr>
<td>Core frequency</td>
<td>2.4GHz</td>
</tr>
<tr>
<td>Number of cores</td>
<td>2</td>
</tr>
<tr>
<td>FPUs</td>
<td>2</td>
</tr>
<tr>
<td>Hw. threads</td>
<td>2</td>
</tr>
<tr>
<td>Hw. threads / core</td>
<td>1</td>
</tr>
<tr>
<td>L1 cache (total)</td>
<td>128K</td>
</tr>
<tr>
<td>L2 cache (total)</td>
<td>3MB</td>
</tr>
<tr>
<td>L1 cache / core</td>
<td>64K</td>
</tr>
<tr>
<td>Memory interface</td>
<td>1x DDR3-1066</td>
</tr>
<tr>
<td>RAM in system</td>
<td>4GB</td>
</tr>
<tr>
<td>Chip technology</td>
<td>45nm</td>
</tr>
<tr>
<td>Chip area</td>
<td>107mm²</td>
</tr>
</tbody>
</table>

† The implementation for the DDR3 interface was incomplete, see section 13.7.1 for details.

Table 13.1: System characteristics.
architecture. This result, repeated over most other benchmarks, confirms that the low-overhead hardware support for concurrency management in the proposed design is able to exploit more software concurrency than traditional software-based approaches.

For larger numbers of cores, the performance then saturates. Various effects cause this limitation, including memory access latencies, overhead of communicating concurrency management events over a larger number of cores, memory bandwidth, load imbalance. Again, a full analysis of the performance characteristics are outside of the scope of our dissertation. Nevertheless, we were particularly excited to find that the performance of the 1-core legacy baseline was matched in multiple benchmarks by less than 32 microthreaded cores, e.g. 8 cores in the running example above. Considering the technology details from table 13.1, this corresponds to a smaller area on chip than 1 core of the selected legacy platform, which was close to the state of the art at the time of this writing. In other words, for this specific benchmark the performance per unit of area is higher with the new architecture. Combined with the observation that the proposed design does not use speculation in any way, contrary to the legacy design, these results suggest that the performance per watt is also higher.

13.2.3.2 Observations from the software engineer’s perspective

The naive rewrite of the sequential C loops (e.g. listing 13.2) as a thread family (e.g. listing 13.4) is sufficient to obtain, after compilation through our tool chain, performance figures close to the hand-optimized assembly code (e.g. more than 85% in the example). The overhead of using SL comes mainly from the absence of global common sub-expression elimination (across thread programs), which forces array base address to be recomputed in every logical thread. In other words, despite our simple and coarse approach to compilation in chapter 6, we could successfully rely on the interface language to expose the performance opportunities of the new architecture.

The preliminary results as to the applicability of the new design to existing software code bases are also encouraging. Indeed, the benchmarks show that code automatically parallelised from C using the partner technology, reported on in [SEM09, SM11] can attain both a higher performance than the legacy baseline within the same chip area budget, and also multi-core performance scalability, cf. e.g. fig. 13.2.

13.3 Separate impact of multi-threading and load balancing

The evaluation activities have revealed that the key parameters to optimize execution efficiency are multi-threaded execution per core and techniques to optimize load balance across cores, and these are mostly orthogonal.

To illustrate this, we use an example program which exposes a heterogeneous workload. The workload is defined by a bulk creation of 40000 logical threads where the amount of work per thread varies irregularly, as illustrated in fig. 13.5.

We show the performance of this workload in the same environment as section 13.2 (table 13.1) in fig. 13.6. We used two implementations that differ in how the logical threads are distributed across cores:

- in the “even” implementation, the logical range is divided into $P$ equal segments where $P$ is the number of cores in the cluster, i.e. core $p$ runs indices $\{\text{start}_p, \text{start}_p+1, \text{start}_p+2, \ldots\}$. This is the straightforward use of the hardware bulk creation process, by requesting a single bulk creation over all cores in the cluster;
13.3. SEPARATE IMPACT OF MULTI-THREADING AND LOAD BALANCING

Figure 13.5: Actual thread sizes in the example heterogeneous workload.

Side note 13.2: Description of the example heterogeneous workload.

Each logical thread $l$ in the range $[0, 40000]$ evaluates the function $m(l)$ defined by

$$
\begin{align*}
    m(l) &= f(x_{\text{start}} + x_{\text{step}} \times (l \mod N) + i \cdot (y_{\text{start}} + y_{\text{step}} \times \lfloor l/N \rfloor)) \\
    f(c) &= \min \{ \lfloor |z_k| \rfloor \cup \{64\} \} \quad \text{with } z_0 = c \text{ and } z_{n+1} = z_n^2 + c \\
    x_{\text{step}} &= (x_{\text{end}} - x_{\text{start}})/N \\
    y_{\text{step}} &= (y_{\text{end}} - y_{\text{start}})/M
\end{align*}
$$

where $(x_{\text{start}}, x_{\text{end}}, y_{\text{start}}, y_{\text{end}})$ are input parameters that define a window over the complex plane, and $(N, M)$ are input parameters that define the discretization of this window. We use:

- $N = M = 200$, hence $N \times M = 40000$ logical threads;
- $x_{\text{start}} = y_{\text{start}} = -2$, $x_{\text{end}} = y_{\text{end}} = 3$.

The function $f$ computes which iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ first escapes the closed disk of radius 2 around the origin, with a maximum of 64 iterations. This is the function typically used to visualize the boundary of the Mandelbrot set [PR86]. We provide the corresponding source code in Appendix K.

Figure 13.6: Performance of the example heterogeneous workload.
• in the “round-robin” implementation, the logical range is distributed in a round-robin fashion over the $P$ cores in the cluster, i.e. core $p$ runs indexes $\{p, p + P, p + 2P \ldots\}$. This triggers bulk creation separately with different logical index ranges on every core of the cluster.

As can be observed in fig. 13.6, the performance of the proposed platform exceeds the reference baseline consistently past 32 cores, and for some parameters beyond 2 cores. This is compatible with the observations from section 13.2.3.1.

Furthermore, fig. 13.6 reveals that the round-robin distribution is radically beneficial to performance. The reason why this is so is exposed more clearly in fig. 13.7: with the even index distribution, the heterogeneity of the workload causes imbalance between cores, whereas the round-robin distribution exploits the local homogeneity of the computational problem to spread the workload more evenly across cores.

This example illustrates the following:

• per-core multithreading is effective at increasing per-core utilization, i.e. instructions per cycle on each core, regardless of load distribution. This can be observed in fig. 13.7b relative to fig. 13.7a and fig. 13.7d relative to fig. 13.7c. While this is a well-known effect for I/O- or memory-bound workloads, this benchmark confirms that fine-grained multithreading is also effective at tolerating latencies of FPU operations, which are handled asynchronously in the proposed architecture.

• the default logical index distribution performed by the hardware bulk creation process, primarily designed for the deployment of regular data-parallelism across cores, i.e. as
13.4. Applicability to irregular functional concurrency

There is limited support in the platform to resolve load imbalance in functionally concurrent programs where the amount of work per sub-problem is dependent on the input. An example was provided by the evaluation of QuickSort using SAC on the proposed platform (cf. side note 13.3). QuickSort uses divide-and-conquer concurrency where the depth of any sub-tree, and thus the complexity of any spatial sub-part of the concurrent workload, is dependent on the values to sort. The design from part I does not provide any hardware support for dynamic load balancing; instead, the QuickSort evaluation carried out by our audience used different explicit placement strategies using our proposed operators from table 11.1.

An example performance graph is given in fig. 13.8; it reports the time to sort 1024 integers using three implementations: one using the current core and the next in the cluster at each divide step (fig. 13.9a); another using the previous and next cores (fig. 13.9b); the last using the upper and lower half of the current cluster at each divide step (fig. 13.9c). Although fig. 13.8 shows relatively good scalability of the last implementation up to 16 cores, the work distribution is still strongly imbalanced, as shown in fig. 13.9d.

In an attempt to minimize such dynamic load imbalances, we jointly co-designed an extension to the hardware bulk creation context with our audience, where the “allocate” message sent to a cluster (cf. section 3.4.2) would travel two times through all cores, one time to select the least used core and the second to actually reserve the thread context.
CHAPTER 13. CORE EVALUATION

Figure 13.8: QuickSort performance on the proposed platform.

Figure reproduced with permission from [SHJ11].

(a) Current/next distribution.

(b) Previous/next distribution.

(c) Lower half/Upper half distribution.

(d) Lower half/Upper half distribution on 64 cores.

The y-axis (vertical) represents the number of logical threads per core. The x-axis represents the code index in the cluster. The z-axis (depth) represents unfolding steps, i.e., time during execution.

Figure 13.9: Different logical thread distributions for QuickSort.

Figures reproduced with permission from [SHJ11].
13.5. OPTIMIZATION OF PERFORMANCE PER WATT

We considered that the higher latency of a two-pass transaction would still be relatively small compared to a software-based load balancing scheme, and would be compensated by a lower load imbalance. To enable the use of this feature in software we introduced the optional keyword “\texttt{sl\_strategy(balanced)}” as a specifier for the constructs \texttt{sl\_create} (Appendix I.5.8.1) and \texttt{sl\_spawn} (section 6.3.5).

Analytically, this \textit{automatic load balancing} feature is only beneficial to performance when the rate of new delegations over the entire local cluster is lower than the bandwidth of the delegation network for the cluster. Otherwise, contention occurs on the delegation network: the latency of each request increases and may not be compensated any more. The applicability of this feature can thus only be increased either by using a coarser concurrency granularity, or increasing the delegation bandwidth.

This feature was thus combined with concurrency throttling in the QuickSort example: the code was modified to use load balancing, and to perform sorting sequentially for sub-lists of less than 10 elements. As fig. 13.10 shows, this combination indeed enables improved performance up to 8 cores (cf. fig. 13.8 for comparison). In this benchmark, beyond 8 cores the increased latency is still not properly compensated by the workload on each core. It is possible to increase this threshold at the cost of more load imbalance.

13.5 Optimization of performance per watt

Another aspect illustrated by fig. 13.6 is the spectrum of possible parameter choices when selecting an implementation to meet a performance constraint: one can either tweak the number of thread contexts per core, the number of cores actually used or the load distribution of logical threads across cores.

When performance constraints are expressed as \textit{real-time deadlines}, e.g. “this computation must complete in less than 1ms,” there may still exist multiple parameters that satisfy the constraint. With the example from fig. 13.6, this specific deadline can be met using e.g. 4 cores with round-robin distribution and 16 threads contexts used per core, or 64 cores with even distribution and 1 thread context per core. We illustrate this diversity in fig. 13.11.

Although intuitively, a selection should favor a smaller number of cores to minimize energy usage, i.e. the parameters of figs. 13.11b to 13.11d over those of fig. 13.11a, the optimal decision strategy may not always be to choose the highest performance per unit of area. For instance, the best choice may depend on the availability of frequency scaling.
Without frequency scaling, parameters that perform with an overall load imbalance may be beneficial as they would allow the system to gate the clock of and power off cores that become unused over time. This would favor choosing e.g. the parameters of fig. 13.11b over those of fig. 13.11c. In contrast, if frequency scaling is available, the configuration of fig. 13.11c can be run at a third of the clock frequency and both configurations of figs. 13.11b to 13.11d may have a similar energy cost. The selection can then be guided by other considerations such as heat dissipation, which would then favor the configuration of fig. 13.11c which better spreads the load than those of figs. 13.11b and 13.11d.

These considerations support our earlier remarks from section 11.3.2: we still lack a performance model which accounts for both configurable core cluster sizes and energy usage by computations. For this reason, a simple on-line resource manager that dynamically places computations based on resource availability and application demands still eludes us—despite, and perhaps regardless of, the availability of hardware primitives for concurrency management.

To summarize, the evaluation confirms that hardware microthreading as an architecture design direction can increase computing density (instructions executed per unit of time and unit of area), including for heterogeneous concurrency, but it does not fundamentally change the problem of making high-level scheduling decisions. If anything, it makes it computationally harder due to the larger amount of concurrency being managed.
13.6 Applicability to throughput-oriented applications

The authors of [TLYL04, YLT05] have introduced NPCryptBench, a benchmark suite to evaluate network processors. We have run unoptimized code for these ciphers and hash algorithms on our reference platform. First the throughput of the unoptimized code for one flow on one microthreaded core is compared against the unoptimized throughput for one flow on one core of the Intel IXP chips ([TLYL04, fig. 4], [YLT05, fig. 3]). Two codes are used on our platform, one purely sequential and one where the inner loop is parallelised. Both are implemented using only our proposed interfaces from chapters 6 and 11. As the
results in fig. 13.12 show, the microthreaded hardware provides a throughput advantage for the more complex AES, SEAL and Blowfish ciphers, whereas the dedicated hardware hash units of the IXP accelerate MD5 and SHA-1. For the other kernels, the microthreaded hardware is slower: with RC5, RC6 and IDEA, a carried dependency serializes execution and minimizes latency tolerance. With RC4, the modified state at each cipher block must be made consistent in memory before the next thread can proceed, which also partly sequentializes execution. Further throughput deviation from the IXP should be considered in light of the frequency difference (1.4GHz for the IXP vs. 1GHz for our platform) and the fact the microthreaded hardware was not designed specifically towards cryptography.

Figure 13.13 shows the scalability of the most popular cryptographic kernels, using the purely sequential, unoptimized code for each stream on our platform and the Level-2 optimized code for the IXP2800 ([TLYL04, fig. 6], [YLT05, fig. 8]). For each sub-cluster size, increasing the number of flows per core increases utilization (fig. 13.14) and thus overall throughput. Throughput is furthermore reliably scalable up to 16 cores. With RC4 and 64 flows on 16 cores the workload reaches the memory bandwidth of the chip; with additional flows, contention on the internal memory network appears, and the utilization is reduced slightly as well as the throughput. The throughput then stabilizes at 96 flows around 1.6Gbps.

13.7 Issues of system bandwidths, design trade-offs

13.7.1 External bandwidth to memory

According to the Amdahl/Case rule of thumb on balanced designs, cited and updated in [GS00], a platform should provision 1 bit per second of external bandwidth for each potential instruction per second. With cores clocked at 1GHz this would imply 1Gbit/s per core. At first sight, this seems matched in the proposed configuration: the internal cache network supports 64-byte transfers at 1GHz, totalling 512Gbits/s internal bandwidth, and uses 4 DDR3-1600 external channels totalling 409Gbits/s external bandwidth, well above the 128Gbits/s implied by the rule of thumb.

Yet while carrying out evaluation activities with our community we did observe experimentally that computation kernels running on 1 to 64 cores, i.e. before fully utilizing the 128 cores available, can saturate the memory bandwidth. We reported on this in [BGH+11]. Further analysis with our peers in charge of the hardware design has revealed the following limiting factors:

- a programming error in the simulation infrastructure: although the DDR interface specification supports pipelining, this feature was not used in the cycle-accurate platform simulation used for evaluation. Instead, this implementation supported only one outstanding request to external memory. The maximum bandwidth for consecutive reads was thus determined by DDR’s read latency (tCL), and not the issue delay (tCCD), yielding a maximum of 34Gbit/s per DDR channel in the reference configuration instead of the expected 102.25Gbit/s;
- even if full pipelining was available, multi-thread interference in Dynamic RAM (DRAM) accesses would still limit bandwidth. This is inherent to the low-level access protocol to DRAM banks: the time to load a row of cells into the row buffer is higher than the time to access different cells within the row buffer. While successive accesses by a single thread can be expected to target addresses within the same row, accesses by
13.7. ISSUES OF SYSTEM BANDWIDTHS, DESIGN TRADE-OFFS

multiple threads are interleaved on the channel and may require in the worst case to switch DRAM rows at each memory operation. This would incur DDR’s row precharge (tRP), activate (tRCD) and minimum open-close latencies (tRAS) at every access, and thus limit the bandwidth for DDR3-1600 channels to 10.5GBit/s per channel.

This last issue is the most severe and has affected all the memory-bound results reported in this chapter and previous academic publications up to 2012. Under high load by heterogeneous multithreaded workloads, the visible bandwidth of 4 DDR3-1600 channels may degrade to 42Gbit/s, well below the 128Gbit/s recommended for a balanced design. To overcome this issue, the system designer may consider that the DRAM access delays only constrain accesses to a single bank. Additional bandwidth can thus be obtained by increasing memory-level parallelism, i.e. the number of visible banks, and expose the address-to-bank mapping to operating software so that it can map different activities to different banks.

This solution was identified in [JYS+12]. However, the authors of this paper explain that increasing memory-level parallelism has a non-trivial cost. With the advent of narrow point-to-point memory interconnects such as FB-DIMM [HV05, GJWJ07] and Intel’s QuickPath Interconnect [Cor09, ZBMS10], it becomes possible to overcome the traditional package pin count limitation and increase the number of separate memory channels; however this comes at the cost of extra latencies and power consumption. Alternatively, one may want to increase the number of independent banks per DRAM module, however the authors estimate that market effects will prevent this opportunity and mandate increasing core count to bank count ratios in future systems. Instead, the authors suggest to use DRAM sub-ranking, which allows the memory controller to load data from different ranks into the same row buffer; however this comes at the cost of extra logic and latency per memory channel.

Future work must thus determine the technology sweet spots that maximize external memory parallelism, and thus the visible bandwidth, at a given logic and energy cost. The memory topology must be exposed at the machine interface and the operating software must use this information to map different software activities to different DRAM banks.

13.7.2 Internal bandwidth

The overall internal bandwidth of the on-chip networks constrains the communication patterns of distributed algorithms and the minimum latencies of SIMD/SPMD operations. As such it is a factor in the maximum performance that can be reached for a given workload.

Any given choice of platform parameters will yield a specific set of network properties. For example, the memory network in the reference configuration has a theoretical point-to-point maximum bandwidth of 512Gbit/s, and the delegation/distribution NoC has a maximum point-to-point bandwidth of 8Gbit/s. Furthermore, any choice of protocol will impact the visible internal bandwidth of the chip. For example, the proposed distributed cache network uses update and eviction messages to propagate stores across caches. These messages reduce the bandwidth available to other messages like loads. In [BGH+11] we discovered that inter-cache management messages can cause a reduction of up to 40% of the visible on-chip memory bandwidth.

We then considered whether this result is a suggestion that the chip designer should provide wider network lanes and higher connectivity, and/or whether new protocols should be designed with more control to software to avoid unnecessary traffic. However, any investment of logic into the network would reduce the maximum number of cores, and thus possibly become detrimental to the performance of compute-bound workloads. This should remind
us of the discussion in [EBSA+11], where the authors argue that for any given many-core chip configuration a significant area of logic will be under-utilized, invisible to most workloads except the few that require it; this is subsequently called **dark silicon**. Addressing this issue may involve the integration of re-configurable logic on chip, where a resource manager on the chip would configure gates towards either extra network lanes (higher communication bandwidth) or cores (higher computation throughput) depending on the workload. We are not aware of any existing research in this direction at the time of this writing, and this issue may constitute a basis for future research.

□ In the mean time, we should highlight here that any **quantitative analysis** of the high-level issues about the internal parameters of a chip cannot occur before specific chip parameters are selected. We can thus argue that the **crystallization** of a platform, such as performed in this book, constitutes a necessary first step towards the development of actual CMPs around hardware microthreading.

### 13.8 Relevance of thread-to-thread synchronizer sharings

We have explained in section 4.3.3.3 that the ability to share physical synchronizers between multiple threads opens the opportunity to daisy-chain logical threads by overlapping the visible window of adjacent thread contexts and spreading the logical indexes in a round-robin fashion.

This feature was originally proposed to explore whether bulk created microthreads are a suitable alternative to dependent sequential loops. The general idea is that expressing a sequence as a network of dependent threads allows dependent threads from a loop to execute in parallel and reduces the need for hardware logic that “discovers” concurrency from the instruction stream. It was implemented in our platform, and we exposed language constructs to use it in chapter 6. An example benefit of this feature can be seen above in fig. 13.12. In these results, a **single stream** of data flows through a cryptography kernel implemented both sequentially and using dependent microthreads for the inner loop. As illustrated in the figure, using dependent microthreads increases the performance of most kernels by 40%-90%.

Yet further analysis by [SEM09] suggest that the feature has only marginal benefits. It only enables performance gains when both the following conditions are met:

- the loop has a constant stride;
- all carried dependencies fit in synchronizers (note that their number is limited due to the substrate’s ISA register naming in the instruction format—our parameters from chapter 8 further limit their number to 6).

Furthermore, the authors of [SM11] propose to rewrite automatically data-parallel workloads that would require a dependent loop in sequential code using a regular dataflow graph that would be efficiently executed using a specialized software scheduler running on multiple microthreaded cores. This transformation has been prototyped successfully in a compiler. As illustrated by fig. 13.2 (“Parallelised C”), this scheme obtains performance figures close to the hand-optimized code.

□ Beyond the software scheduler of [SM11], we have identified other general ways to organize partially sequential computations between microthreads using only **existing features of the design**:
13.8. RELEVANCE OF THREAD-TO-THREAD SHARINGS

- plain sequence between logical threads on each core can be requested by setting the “block size” parameter from section 4.3.2.2 to 1;
- any reduction using commutative and associative operations on one core can be implemented using only the “global” synchronizers shared by all thread contexts participating in a bulk creation:
  - simple operations that require only one machine instruction can be used with a global synchronizer as both input and output operand. The dataflow scheduler will automatically serialize all such instructions in the pipeline, possibly out of logical index order;
  - complex operations that either require multiple machine instructions, or involve updates to memory which must appear atomic, can be enclosed by a transaction started with a “clear” operation to one of the global synchronizers, and ended with a “set” operation to the same synchronizer. The clear operation both reads from its operand and sets its dataflow state to “empty.” Once the first clear executes, all further clears from other threads will suspend until the synchronizer is set again. The set is then performed by the thread that has executed the first clear at the end of its transaction. This wakes up another thread, which is given a chance to run “clear” and open a new transaction, and so on.
- any reduction using associative operations on multiple cores can be implemented using successive bulk creations of thread functions performing a parallel prefix scan [LF80, SHZO07] of the data to reduce, or distribute local reductions as described in section 11.2.4;
- efficient multi-core reductions using multithreading on each core for latency tolerance can be achieved by combining the techniques above.

We have tested these opportunities and confirm anecdotally that they yield high core utilization. Moreover, using global synchronizers for reductions further reduces register file utilization, since the global synchronizers are only allocated once. To simplify the use of global synchronizers for reductions in programs, we further suggest to introduce Cilk’s hyperobjects [Lei09] as a low-level language feature in our interface language, upon which further abstractions can be constructed; we are planning this implementation as future work.

To summarize, we have not yet found a decisive benefit of the “shared synchronizers” pattern; instead we observed that its intended use cases can be addressed using other, simpler features. Also, as illustrated in figs. 13.13 and 13.14, interleaving different purely sequential activities on the same cores provides another means to maximize utilization of each core. These observations may indicate that the “shared synchronizers” feature was an instance of premature optimization. We could not conclusively support the need for this feature; future work may even suggest to remove it from implementations to further simplify the microthreaded cores and their conceptual model. Our careful separation of its description in chapter 4, which isolates this feature from the more interesting aspects of the architecture, was an intentional step in this direction.
Summary

In this chapter we have shown that the provided platform support was sufficient to carry out evaluation via reduced, yet fully functional benchmarks. Regardless of the specific performance figures, the fact that evaluation was possible using traditional engineering workflows is a sign that the platform “looks and feels” like traditional platforms sufficiently for integration in existing evaluation methodologies. Furthermore, by looking at the evaluation results we show that the proposed tool chain was appropriate to utilize efficiently the processing abilities of the new architecture for most data-parallel workloads, and obtain higher performance densities than contemporary chip architectures. As predicted in section 5.3, our technology empowered both our community and ourselves to step out of the engineering effort and gain high-level insight over issues of concurrency management and chip design, briefly exposed in this chapter.