On the compilation of a parallel language targeting the self-adaptive virtual processor

Bernard, T.A.M.

Citation for published version (APA):
Enschede: Print partners Ipskamp

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: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Discussion and conclusion

African Proverb

For tomorrow belongs to the people who prepare for it today.

In this dissertation, we have addressed the problem of integrating concurrency constructs (in both source and target languages) and assumptions (from the execution model) into an imperative-language compilation software system. As technical contribution, we have developed a working compiler for the μTC language targeting the Microgrid architecture. The compiler is extended with the properties of the SVP concurrent execution model. This work has served as a basis to investigate several aspects of concurrency for programming multicore architectures. In particular, this work has focused on the integration of concurrency assumptions in a non-concurrency-aware transforming software system that bridges the two SVP implementations. This chapter reviews the different problems and results presented in this thesis and summarizes their implications.

8.1 Thesis overview

8.1.1 Programming multicore architectures with SVP

Chapter 2 has reported an overview of different layers of existing concurrent software systems: execution models, parallel architectures, concurrent programming languages, and compilers. We have discussed the context of our research when we have looked at how to program multicore architectures. We have then described the candidate used as a case study throughout this thesis in Chapter 3, the Self-Adaptive Virtual Processor execution model (SVP). This model
was resolved into software and hardware implementations with SVP properties. In Chapter 4, we have looked at compilation techniques to allow the bridge between a concurrency-oriented language to a concurrent target platform. Besides the clashes in compiler optimizations reported in Chapter 5, we have made a working compiler based on an existing imperative-language compiler framework. Chapter 6 has presented the challenges we have encountered during this research to embed native concurrency idioms within a sequential-oriented framework. Chapter 7 has evaluated the SVP extension to an imperative-language compiler with scientific problems. In this chapter, we use this experience to refine the current status of compiler research for concurrent systems. The multicore programming menace is still present and the work to perform on tools to fit the concurrent systems is far from being finished.

8.1.2 SVP compilation

The motivation of integrating SVP properties into an existing imperative language is the opportunity to reuse existing compiler technology that is proven to work. Moreover, it has been also a way to produce results within the time constraints of this research. The other option would have been to implement a compiler from scratch; this was not relevant with our research goals:

- compilation of programs to the SVP hardware implementation,
- investigate the impact of concurrent programs on existing development tools,
- provide solutions to support concurrent assumptions,
- expose dangers on relevant areas to monitor in compiler research.

8.2 Limitations

8.2.1 General limitations

Compiler research comprises a large scope of topics that we could not cover within the time allocated to this thesis. We first made restrictions with the choice to embed concurrency directly into the compiler’s infrastructure. In addition to that, the choice of the compiler as GCC constrained the development in a Test-Driven Development (TDD) scheme which consumes time and resource to obtain a working compiler. Compilers, as middle-men, are positioned in between two worlds (software and hardware) with constant pressures to deliver more efficient compiled programs. With the appearance of multicore architectures and lack of concurrent programming standards, compilers have the job to take advantage of sequential programs (with and without concurrency annotations in the code) and to discover parallelism in the code or just to
compile with an optimizing objective. The same happened in the context of our research where the compiler development was pressurized by frequent changes of the target platform. We then observed that simple changes in the target’s instruction set have considerable impact on compiler modifications to adapt to these changes. It becomes clear to say that a close collaboration between compiler architects and architecture designers is prerequisite for such research in software systems. Here, we made the assumption that the software side is a passive actor of the system. Software standards indeed do not evolve as fast as architectures. Developers are skeptical of new standards and are reluctant to learn new programming techniques. Consequently, the pressure of finding and handling concurrency goes to the developer tools, such as compilers.

During this research, we encountered challenges due to changes in a moving target platform and its instruction set, including changes in the programming language. These impacted considerably on compiler development which is hard to quantify. This resides mainly in our choice to expose concurrency explicitly in the source language with \( \mu \)TC \cite{64} and in the target instruction set with the Microgrid \cite{61}. \( \mu \)TC uses parts of the C-language properties and extends them with new constructs to support SVP properties. The Microgrid uses a DRISC approach \cite{43} and extends a conventional instruction set with new instructions to support SVP properties. In theory, \( \mu \)TC constructs map directly onto Microgrid instructions with regard to SVP assumptions. However, in practice, SVP semantics break conventional assumptions especially in compiler optimizations and C-language compilation schemes. The latter requires extensive support in a compiler’s infrastructure to preserve the two principles of compilation:

1. The compiler must preserve the semantics of the input program being compiled.
2. The compiler must improve the input program in some perceptible way (in the case of an optimizing compiler).

### 8.2.2 Current compiler limitations

The SVP compiler has proven to work but it has limitations. The first limitation is related to the implementation of the thread local storage (TLS) which is concurrent stack management for multicore architecture, investigated in \cite{91}. Consequently, spilling techniques are not yet implemented with the compiler which limits the amount of parameters that can be passed at thread creation (cf. Section 3.3.7). The compiler cannot reuse physically shared registers via spills: a spill itself would require a read from the register, thus forcing synchronization with the previous thread and reducing concurrency.

Due to time constraints, regular C function calls are not yet implemented. The implication of supporting this in the back-end requires the work described in Section 4.2 and combines it with a complex back-end. This back-end forks
into two parts: the conventional sequential calling convention and the concurrency creating convention. The thread creation mechanism reuses parts of the call mechanism and its calling convention. The back-end therefore needs to support different types of calling conventions which is not the case in the compiler’s infrastructure.

As mentioned in Section 6.3.2, the Microgrid implementation employs variable (dynamically-sized) register windows for each thread function. In other words, this removes the long standing assumption that the size of the architectural register window is a constant for all programs. The register allocator requires hard descriptions of the target’s memory before compiler generation. Supporting variable register windows requires big changes in the concepts of register allocation; the calling convention and memory description are not static anymore. To render feasible the compilation of programs within the time of this research, we have fixed the register windows for the Microgrid implementation. Nonetheless, this directly impacts the code quality for register windows which are not optimal in size. In other words, the size of register windows can sometimes not be the smallest possible.

8.3 Future work

8.3.1 Possible improvements

This SVP working compiler has room for improvements. First, the verification and evaluation steps will be pushed further with the results of the AppleCORE project [90]. This project investigates the application of SVP concurrency in larger benchmarks and provides results for industry-standard benchmarks. The SVP compiler will then have a larger base for discovering further sleeping compiler defects than those already discovered. Moreover, the SVP model and its implementation are evolving continuously; the compiler will have to follow this evolution. The last major improvement in the SVP hardware implementation is the change of the create mechanism from a ‘pull’ scheme to a ‘push’ one. This is discussed in Section 6.4.1. This will impact considerably on the back-end complexity; there is no bound on variable initialization (parameter passing) anymore, between the created family and its parent beyond the create action. The compiler does not have to preserve locks on resources for parameter passing and the register allocator becomes simpler to design and implement. The main reason for the change is the recent addition of continuation create in the SVP model to support, amongst other things, the implementation of system services in an operating system. In a continuation create, a family of threads is created with no existing link with its creator beyond its issue point. This SVP property is required for further investigation in operating system and resource management for SVP.

Beyond SVP changes, the SVP compiler requires some more work and investigation for advanced and aggressive optimizations used with GCC optimiza-
tion flags -O2 and -O3. **Instruction reordering**, which reorders instructions and breaks SVP dependency chains between operations; a shared communication channel may be written before being read for instance. Other optimizations would require more investigation, especially SVP-related optimizations that could occur at the middle-end level. Optimizing the concurrency tree of a program is a potential direction, with rebalancing the workload of thread families by reshaping/restructuring block size and other family parameters. To obtain this, we could introduce feedback from the target in order to report to the compiler with some metrics the behavior of the concurrency tree during execution. Based on these metrics, the optimizing algorithm may reshape the concurrency tree to obtain a better balance of workload throughout the Microgrid. Existing feedback compilation using GCC is already in the market with MILEPOST [92] that uses the GCC plug-in ICI mechanism [86]. Moreover, target’s resources are static at design-time; but, they may change at execution-time. The compiler can introduce variables and code sections at compile-time to permit other execution paths. These code sections and variables will be interpreted by the run-time system which sets and optimizes based on this special information. This optimization can be done without feedback but simply with known target parameters which be known at run-time.

### 8.3.2 Future perspectives of Compiler Research

This research has shown other reflections than the ones related to SVP compiler research. The emphasis on compiler technology is getting more important with the presence of multicore architectures. The compiler does not only have to transform a program correctly and to improve it, but it has to discover and take advantage of concurrency in the input program. Developers may use hints to expose concurrency in a program with various programming paradigms; compilers have to evolve as well and need to be recast with new compiler models to become more flexible and better compartmentalized. Most compilers suffer from a sequential compiler model with a fixed infrastructure (understand the sequence of front-end, middle-end, and back-end) which is not flexible for the addition of new components or concepts. There is therefore a need for more flexibility and modularity in next generation of compilers. In that sense, the CoSy framework [79] is already in ahead of its time. However, although the GCC framework struggles with this sequential compiler model, efforts are currently being made to modernize its compiler model: the ICI plug-in initiative [86] and with the Modular-GCC initiative [87].

The next generations of imperative-language compilers will have to encapsulate an abstract representation good enough to model concurrency in the compiler’s internals, and therefore to avoid concurrency discrepancies. Compiler optimizations are a major problem when dealing with concurrency properties; it becomes clear that the compiler wall will be hit if compiler models are not recast as a whole new system with the embedding of concurrency. This is an entire research area where sleeping bugs in existing sequential programs will
be exposed when they encounter concurrency. In the end, from the user perspective, the developer community is too conservative to change their way of thinking and designing applications. Therefore, development tools, such as compilers, will have to do the job and relieve stress from them by handling concurrency in input programs.

## 8.4 Conclusions

To conclude this thesis work, we have addressed the problem of integrating concurrency constructs and assumptions into an imperative-language compilation software system. The context of this thesis work is the programmability (or programming) of multicore architectures where the SVP execution model is a potential candidate and used as a basis for this study. The SVP execution model provides a solution as a whole with different layers (and tools), in which each has its own responsibility and the adequate separation of concerns to manage concurrency. This thesis has focused on specific areas:

1. presentation of the SVP execution model,
2. compilation of a concurrency-oriented language to a multicore architecture,
3. investigation of interactions of concurrency assumptions in conventional compiler optimizations,
4. exposition of challenges with SVP compilation and concurrency matters.

We have exposed that integrating concurrency assumptions of the SVP execution model is technically possible and the working SVP compiler stands as the technical contribution of this thesis. However, we have also drawn up limits of such an approach: embedding concurrency constructs into an imperative paradigm. The first distinct impact is the problem with the tools that are designed in a way that will force the reengineering as a whole to get dedicated concurrency-aware tools. These compilers will be modular in their design to enable flexibility of retargeting to various architectures. Moreover, it will still take some time until concurrency standards appear and are approved by the developer community. Until that time, efforts in compilation will have to be performed to provide production-quality solutions. The reason for this reengineering is mainly due to clashes related to concurrency assumptions that break optimization algorithms and compilation schemes. Therefore, there is a need for dedicated solutions for the matter of programming multicore architecture.