A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs

Ettore Speziale* and Michele Tartara
Dipartimento di Elettronica e Informazione (DEI)
Politecnico di Milano
via G. Ponzio 34/5, 20133 Milano, Italy
Email: {speziale,mtartara}@elet.polimi.it

Abstract

This paper describes a dynamic and lightweight compiler able to guide the execution of highly dynamic parallel programs at runtime without the need for a full-fledged Just-In-Time compiler. The proposed approach allows us to apply profitable optimizations that could not be used at compile-time due to lack of sufficient information. It also enables us to optimize code fragments multiple times, depending on the specific conditions of the execution environment at runtime, while limiting the performance overhead to a negligible amount.

1 Introduction

Trends in computer science show an ongoing shift of paradigm, from *sequential* to *parallel*, because of the inability to further increase clock rates due to technological and thermal issues [6], and for exploiting the increased transistor density guaranteed by Moore's law [28].

With multi-/many-core processors, the offered parallelism evolved from an *implicit* form (*e.g.:* Out-of-Order execution [36]) to an *explicit* form, where processing elements are directly controlled by programmers.

From an architectural perspective, this allows simplified processor design (by removing power- and area-hungry components, like branch predictors [39]) and freeing up resources to increase the number of processing elements. This approach has been exploited in GPGPU designs [30], where there is a large number of processing elements very similar to an ALU. This leads to an increased importance of the compiler: simpler processors are less able to reschedule instructions, therefore the compiler has to be smart at defining their execution order.

The push toward explicit parallelism influences programming models, compilers and runtimes. Research works has shown that no known technique can deal with all the available parallel architectures and challenges [2]. All the explicit parallel programming models push programmers to define elementary units of work, called either *task*s [12], or *parallel iteration*s [31] or *work item*s [18], and let the compiler and/or the runtime schedule them according to their mutual dependencies.

In this context, compilers have been used in a classical way: they perform translation to machine code, that is later executed. Leveraging on the structure of the input language, some compilers perform aggressive optimizations, such as work-items pre-scheduling [17, 35]. The generated code is always oblivious to the existence of a compiler, and the compiler, even when it is a JIT, does not provide any kind of "service" to the program.

This work presents a more dynamic approach to compilation. We generate code meant to interact with the compiler during execution, to exploit dynamically available information to optimize the code on-the-fly, without the burden of a full-fledged JIT system. In our approach the compiler works together with a micro-thread scheduler. *Pipeline stalls* in micro-threads containing the program code can be used to execute the compiler optimizers, thus *minimizing compiler overhead* at runtime.

The main motivation for pursuing this approach is code optimization: by running the compiler during code execution, more information is available, enabling more precise optimizations.

As a side effect, our approach would benefit software deployed in binary-only form, meant to run on many different hardware configurations (*e.g.:* binary packages used by Linux distributions). Allowing the program to customize itself without being compiled from a bytecode form at deploy-time and without the need for a Just-In-Time compiler (that is time- and resource-consuming at runtime) could be useful in a variety of scenarios.

Section 2 of this paper deals with existing approaches to runtime compilation and code modification. Section 3 introduces our approach to perform runtime compilation using lightweight compilation micro-threads and runtime

*This author was supported in part by a grant from ST Microelectronics.

scheduling. Section 4 discusses scenarios where our technique can be useful and Section 5 concludes.

2 Related Work

The problem of adapting programs to the runtime environment and to the specific set of data they are working on has been tackled in many ways, mostly related to the concept of dynamic compilation, also known as Just-In-Time (JIT) [3]. According to this approach, parts of a program are compiled while the program itself is being run, when more information is available than at compile time and can be used to perform further optimizations.

One of the first works on JIT systems [15] deals with the fundamental questions of JIT: determining what code should be optimized, when, and which optimizations should be used. We deal with similar questions, but we decide at compile time what code to optimize and which optimizations to apply, and postpone to runtime the task of answering "when" and "how" to optimize the code.

JIT compilation introduces an overhead in execution time because it causes the program to be idle while waiting for the new machine code. Considering that most programs spend the majority of time executing a minority of code [20], two papers [8, 10] independently proposed the approach called *mixed code*, where most of the code is interpreted and only the frequently executed part is identified, compiled and optimized at runtime.

Some works [16, 22] exploit multi-core processors to hide compilation latency: the compiler is run in a different thread and uses heuristics to predict the next method to compile before it is actually needed by the program. State of the art implementation can be found in [21].

All JIT-related works assume a compiler is running alongside the program. This causes a big overhead because of the memory and the time it takes to compile a new, optimized version of the code. On the other hand, the approach we propose does not need a full-fledged compiler running alongside the program. It only applies lightweight transformations to code specifically generated at compile-time to allow it, thus requiring much less resources, while being only slightly less flexible than a full-fledged JIT compiler.

Staged compilation is another low-overhead approach. It splits the compilation in a static and a dynamic stage. The static one compiles "templates", building blocks for the dynamic stage that connects them and fills the holes left by the static stage with run-time values [27].

An example of a state of the art JIT compiler is the HotSpot Java Virtual Machine [21, 32], that uses adaptive optimization on top of a mixed code approach, with continuous monitoring and profiling of the program during its execution. It performs non-conservative optimizations, such as inlining frequently called virtual methods.

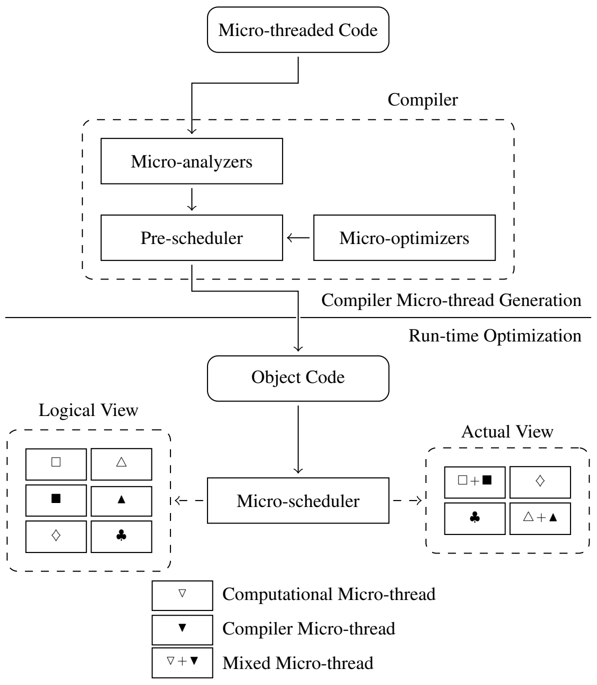


Figure 1: proposed compilation/execution pipeline. Micro-threaded code is analyzed to detect profitable runtime optimizations. Compiler micro-threads (filled-in symbols) are built and possibly merged with computational micro-threads (empty symbols), generating mixed micro-threads (both symbols)

To deal with Java's dynamic class loading it uses *dynamic de-optimization*. When the assumptions that led to a non-conservative optimization become false, the code is de-optimized back to a safe version, and then new optimizations are applied.

HotSpot supports two different compilers, namely the "Client" one and the "Server" one. The HotSpot Client Compiler [21] is focused on optimizing client applications, where the responsivity of the application is more important than deep optimization. The HotSpot Server Compiler [32] aims at optimizing the server applications, where it is worth spending more time compiling parts of the application. Therefore, the compiler features all the classic optimizations, as well as some Java specific ones.

A different approach to adapting programs to the runtime environment is *self-modifying code*. Von Neumann architectures [38] represent code in the same way as data. Therefore, a program is able to modify its own code while running, changing its own behavior. The main drawback of self-modifying code is the difficulty for many programmers to understand, write and maintain such code. Self-modifying code is used in [26] to write an operating system kernel and in Knuth's MIX architecture [19] for the subroutine calling convention.

3 Proposed Approach

In this paper we present a new kind of compiler optimizations, able to adapt to highly dynamic execution environments without adding excessive overhead at runtime.

Optimizations built according to our approach are divided in two phases, one to be executed at compile time and one at runtime. The runtime phase is extremely lightweight and is assigned the task of modifying the program to actually apply the optimization according to the current state of the execution environment, whereas the compile-time phase has to generate the machine code of the program in such a way to allow this to happen

The need for offloading most of the optimization-related computation on the static compiler has already been assessed by other works, such as [29]. Another example of cooperation between compiler and runtime can be found in [14] for GPUs.

With respect to the traditional static/dynamic compilation flow, where compilation and execution phases are clearly separated, we have to face two specific issues.

Expected profitability: not all optimizations have to be delayed at runtime. We aim at applying an optimization at runtime only if *there are no sufficient information* to apply it at compile-time and a *considerable improvement* is expected. At the same time, since code is generated at compile-time, we free the runtime environment from the burden of applying trivial but needed optimizations, such as copy propagation, that a traditional JIT approach has to perform during program execution.

Moreover, delaying at runtime all applicable optimizations is not feasible, because we aim at keeping a lower overhead with respect to traditional JITs. This naturally leads to a careful selection of which optimizations to delay, based on their expected profitability.

Compiler interference: runtime application of optimizations leads to possible conflicts between optimizers and the running optimized code. This happens because there is no strong separation between the compiling and running phases of the program. To guarantee consistency, it is necessary to coordinate optimization and execution of the code.

To handle these issues, we define a model that allows us to detect, handle and apply profitable optimizations. We represent the program using a set of micro-threads (similar to those described in [11, 12, 18]). Part of these micro-threads are defined by the programmer or by the compiler and contain the code of the program being written. We call them *computational micro-threads*. The remaining micro-threads are called *compiler micro-threads*. They are generated by the compiler and contain the code that is able to apply optimizations at runtime.

Each compiler micro-thread is associated to a computational one, and manipulates one of its *optimizable regions*, that are the sections of the code of a computational micro-thread that can be modified by a runtime optimization. The dual of an optimizable region is an *optimizing region*. It is defined as all the code of a computational thread that is not part of the corresponding optimizable region. The optimizing region is the region where the optimizer micro-thread can safely run concurrently with the computational micro-thread to apply its optimization.

3.1 Compilation/Execution Pipeline

With reference to Figure 1, our compilation approach is split into two parts: *generation of compiler micro-threads* and *runtime optimization*.

The first step is intended to be part of a static compilation pipeline, and its goal is preparing the code to be optimized at runtime. We want to consider only optimizations that cannot be applied at compile-time, so this step should be run after standard compiler optimizations. First of all, it has to look at the input code to find candidate applicable runtime optimizations. It is not possible to apply all optimizations, because interferences between them are possible. Therefore, they must be scored with respect to the expected profitability. Then, the model based on optimizable/optimizing regions allows us to represent such interferences on the computational micro-thread control flow graph. A pre-scheduler analyzes the interferences and selects the best optimizations. Finally, the corresponding compiler micro-threads are generated from a library of *micro-optimizers*. For each computational micro-thread, multiple compiler micro-threads can be generated, one for each optimization.

It is worth noting that the micro-threaded model is a purely logical one: we aim at minimizing runtime overhead, therefore if the system is implemented on a computing architecture with high costs of inter-thread communication the micro-threads can be multiplexed into a single mixed micro-thread. To do this, the pre-scheduler analyzes the computational micro-thread and compiler micro-threads, and schedules in a single mixed micro-thread the instructions from both, according to constraints imposed by optimizable and optimizing regions. Merging different micro-threads together was proven to be effective for scheduling Single Instructions Multiple Threads programs [23, 24, 35]. In our approach, micro-threads are not homogeneous, but we think that similar techniques have to be used to limit as much as possible the overhead of runtime optimizations.

The output of the pre-scheduler is a set of threads containing micro-threaded code intended to be run by a runtime micro-scheduler. From the logical point of view, the runtime scheduler has to manage both computational and compiler micro-threads, but, due to pre-scheduling, it actually has to manage mixed micro-threads too: therefore, some of the micro-threads need synchronization, whereas some other micro-threads have already been merged by the pre-scheduler, thus eliminating the need for explicit synchronization.

3.2 Run-time Optimization

The compiler micro-threads have to change the code of their associated computational micro-thread to optimize it. This can be done explicitly, using *self-modifying code*, or implicitly, using *branch tables*.

The compiler micro-thread is generated together with the optimizable region code it is associated to. Indeed, knowing the layout of the optimizable region, it is possible to generate instructions performing binary rewriting at runtime, without influencing other regions of code of the computational micro-thread.

The strength of self-modifying code is the ability to generate the most suitable instructions for a given optimizable region. However, the cost of code morphing is considerable. An entire region of code must be rewritten. This requires editing the memory locations that store the optimizable region. Moreover, if the code is shared by multiple micro-threads, code cannot always be modified: the conditions triggering runtime optimization for a given micro-thread could be not valid for the others. Despite these limitations, self-modifying code can be an effective optimization strategy, if exploited for highly profitable optimizations, like inner loop vectorization.

A branch table, on the other hand, is a collection of unconditional jumps to different locations. At runtime, an index is used to select where to jump to. It can be implemented using different techniques, and is used to translate switch statements or to implement virtual tables. In our context, branch tables enable compiler micro-threads to change the execution flow of the associated computational micro-thread without changing its code. When our logical model is implemented, an optimizable region should be represented as a collection of sub-regions linked using branch table-based jumps. Compiler micro-threads just have to modify the indices used to select the active jump in branch tables, thus implicitly modifying the control flow graph of the computational micro-thread.

With respect to self-modifying code, branch tables impose less runtime overhead, since applying an optimization simply amounts to setting a set of indices. On the other hand, all the possible fragments of code used to optimize the region need to be generated at compile-time. The low runtime overhead makes this strategy suitable for highly dynamic scenarios, where the compiler micro-thread has to modify the execution flow more often.

To trigger an optimization, compiler micro-threads must observe the state of the associated computational micro-thread. If an optimization was postponed at runtime because the value of a variable was unknown at compile-time, the observed state will surely include that variable as one of the elements to be considered to decide when and how to apply the optimization at runtime.

It is worth noting that our approach enables a wide range of runtime optimizations. We use branch tables to restructure the execution flow and, where this is not sufficient, we also allow code morphing to apply deeper modifications. The use of branch tables should not be perceived as just a static branch prediction [5], since it is not performed statically, but is dynamically changed every time it is needed, as a result of modifications in the execution environment.

The strong relationship between computational and compiler micro-threads motivate us to emphasize the importance of having an effective and efficient prescheduler. Data related to a computational micro-thread must be collected and analyzed by the corresponding compiler micro-threads. Moreover, compiler micro-threads change the behaviour of the computational micro-thread. By scheduling the different micro-threads together we aim at avoiding communication delays between them. This guarantees deterministic interactions between micro-threads, as well as high performance. Even if it is strongly discouraged, our proposal does not prevent scheduling compiler micro-threads independently. However, in this case it is required to consider explicit synchronization between micro-threads, possibly exploiting weak memory consistency models [1] to limit communication overhead.

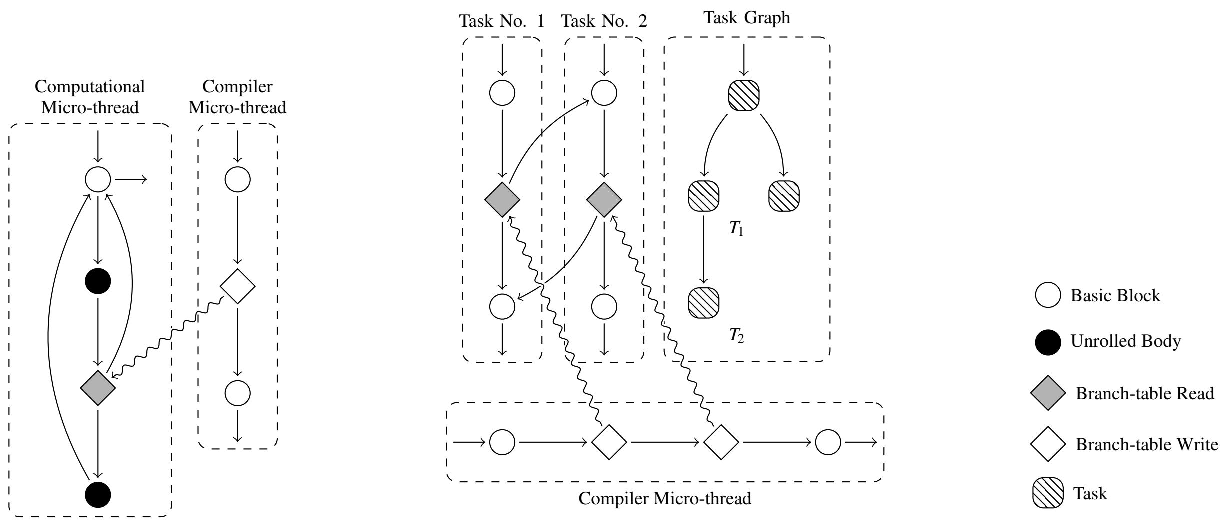
The authors of [25] observe that current production-quality compilers have issues with vectorization because the required analyses, such as interprocedural alias analysis, are not available. Such an analysis is really hard to implement at compile time, but pointers can be disambiguated at runtime. This further supports the need for splitting the compilation effort between compile-time and runtime, as allowed by our approach.

4 Foreseen Applications

In this section we present two examples of optimization that would benefit from our approach. Figure 2 gives a brief overview.

4.1 Adaptive Loop Unrolling

The classic loop unrolling optimizations [4] can lead to improved, unaffected or worsened execution times depending on whether the right unroll factor is chosen [7, 9]. This is a challenging task, requiring good



(a) adaptive loop unrolling (b) dynamic task fusion (c) legend

Figure 2: graphical representation of two foreseen applications of our proposed approach

knowledge of the target architecture [33]. In most cases, this is only available at runtime, and is exploited using a JIT compiler. Unfortunately, JITs are really heavyweight and time consuming.

With our approach, we estimate a maximum sensible unrolling factor k at compile-time. We unroll the code of the loop k times and insert a branch table read between each pair of unrolled loop bodies, as in Figure 2(a). This is the optimizable region. At runtime, the compiler micro-thread determines the best unrolling factor $n \leq k$ (according to the size of caches, the number of required iterations, etc.) and modifies the n-th branch table read so that it jumps back to the loop header, and all the other ones so that they either jump to the next instruction, or are substituted by `nop` instructions.

This approach is much lighter that a full-fledged JIT, but it does not enable the application of further optimizations on the unrolled code. However, if the underlying architecture is micro-programmed, the machine code will be rewritten and optimized by the hardware, making our code comparable to that unrolled by a JIT.

4.2 Dynamic Task Fusion

Task based data-flow programming models have been proven to be an attractive way to tackle some parallel applications [34]: tasks are generated on the fly, thus they require the use of a runtime scheduler to select and start them according to data and control dependencies. Therefore, after each task finishes executing, control has to return to the scheduler so that it can start the next task.

Using our approach, we can define an optimizable region just before the end of the machine code of each task, made of just a branch table read. As shown in Figure 2(b), at runtime, a compiler micro-thread supports the scheduler: it observes the state of the system and modifies the corresponding branch table to have it point to the beginning of the code of the next ready task. Therefore, tasks can be executed continuously, without the overhead of reaching back to the scheduler at the end of each of them. The modification of the branch table takes place as soon as the compiler micro-thread is aware of the next ready task, therefore the current and the next task will be executed one immediately after the other, as if fused together. Some call to the scheduler will still be needed, for example in order to mark a task as finished, unlocking the depending ones.

When the task graph is known at compile time, more aggressive optimizations can be performed [13]. Our approach does not allow this, but it limits the scheduling overhead that arises when inter-dependent tasks have to be executed (as tackled in [37]) and handles highly dynamic applications where the task graph is known only at runtime, even if the code is generated at compile-time.

5 Concluding Remarks and Future Work

In this paper we presented a novel lightweight approach to optimize highly dynamic parallel programs, based on the use of compiler micro-threads that modify the running program at runtime, adapting it to the current environment. We described some optimizations that could implemented using our methodology, to show that is general enough to be applied to a wide variety of algorithms. At the same time, though, it does not need to be completely general-purpose, since it is not meant to completely replace other techniques, such as static optimization or JIT compilation.

We are currently planning the implementation of our compilation toolchain in order to conduct an extensive and accurate experimental campaign.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. P. Lester, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[3] J. Aycock. A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[4] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[5] T. Ball and J. R. Larus. Branch Prediction For Free. In *PLDI*, pages 300–313, 1993.

[6] S. Borkar. Thousand Core Chips – A Technology Perspective. In *DAC*, pages 746–749, 2007.

[7] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, 1994.

[8] R. J. Dakin and P. C. Poole. A Mixed Code Approach. *Comput. J.*, 16(3):219–222, 1973.

[9] J. W. Davidson and S. Jinturkar. Aggressive Loop Unrolling in a Retargetable Optimizing Compiler. In *CC*, pages 59–73, 1996.

[10] J. L. Dawson. Combining Interpretive Code with Machine Code. *Comput. J.*, 16(3):216–219, 1973.

[11] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[13] M. I. Gordon, W. Thies, and S. P. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS*, pages 151–162, 2006.

[14] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. S. M. Goh. Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs. In *IPDPS*, pages 467–478, 2011.

[15] G. J. Hansen. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. PhD thesis, 1974.

[16] U. Hölzle and D. Ungar. A Third-Generation SELF Implementation: Reconsiling Responsiveness with Performance. In *OOPSLA*, pages 229–243, 1994.

[17] R. Karrenberg and S. Hack. Whole-function Vectorization. In *CGO*, pages 141–150, 2011.

[18] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, 2010.

[19] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[20] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[21] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot^{TM}Client Compiler for Java 6. *TACO*, 5(1), 2008.

[22] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.

[23] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee. An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence. In *PACT*, pages 56–67, 2011.

[24] J. Lee, J. Kim, S. Seo, S. Kim, J.-H. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. An OpenCL Framework for Heterogeneous Multicores with Local Memory. In *PACT*, pages 193–204, 2010.

[25] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An Evaluation of Vectorizing Compilers. In *PACT*, pages 372–382, 2011.

[26] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, 1992.

[27] M. Mock, C. Chambers, and S. J. Eggers. Calpa: a Tool for Automating Selective Dynamic Compilation. In *MICRO*, pages 291–302, 2000.

[28] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.

[29] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *CGO*, pages 151–160, 2011.

[30] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, 2009.

[31] OpenMP Architecture Review Board. *OpenMP Application Program Interface, version 3.0*, 2008.

[32] M. Paleczny, C. A. Vick, and C. Click. The Java HotSpot^{TM}Server Compiler. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[33] V. Sarkar. Optimized Unrolling of Nested Loops. In *ICS*, pages 153–166, 2000.

[34] F. Song, A. YarKhan, and J. Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-memory Multicore Systems. In *SC*, 2009.

[35] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO*, pages 111–119, 2010.

[36] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1), 1967.

[37] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A Unified Scheduler for Recursive and Task Dataflow Parallelism. In *PACT*, pages 1–11, 2011.

[38] J. von Neumann. First Draft of a Report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4), 1993.

[39] T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In *MICRO*, pages 51–61, 1991.