Processor Virtualization and Split Compilation for Heterogeneous Multicore Embedded Systems

Albert Cohen, Erven Rohou

Processor Virtualization and Split Compilation for Heterogeneous Multicore Embedded Systems

Albert Cohen
INRIA Saclay – Île-de-France and Paris-Sud 11 University, France, and HiPEAC network
Albert.Cohen@inria.fr

Erven Rohou
INRIA Rennes – Bretagne Atlantique, France, and HiPEAC network
Erven.Rohou@inria.fr

ABSTRACT

Embedded multiprocessors have always been heterogeneous, driven by the power-efficiency and compute-density of hardware specialization. We aim to achieve portability and sustained performance of complete applications, leveraging diverse programmable cores. We combine instruction-set virtualization with just-in-time compilation, compiling C, C++ and managed languages to a target-independent intermediate language, maximizing the information flow between compilation steps in a split optimization process.

Categories and Subject Descriptors

D.3.4 [**Programming Langugages**]: Compilers

General Terms

Performance, Languages, Algorithms

Keywords

Heterogeneous multicore, virtualization, bytecode language, annotations, split compilation, portable performance, specialization, vectorization, back-end optimization.

1. MOTIVATION AND GOALS

Embedded systems have always been heterogeneous multicores. In many cases, the hard constraints in terms of cost, area, performance, power consumption and real-time simply make a general-purpose processor unfit for the task. With the exception of the host microcontroller, hardware components have been dominated by dedicated accelerators. However, the non-recurring engineering cost of chip design, and the exponential increase of application size and variety push for programmable systems targeting wider application domains.

In addition, each platform provider pushes its own solution, typically very different from the competition's offers; evolutions are often not backward compatible. The end result is an extreme fragmentation of the embedded market which is difficult to handle from a software productivity perspective. Indeed, independent software vendors for embedded systems must often deal with tens (even hundreds, in extreme cases) of combinations of target instruction sets, microarchitectures, toolchains and operating systems. They must restrict their developments to niche domains, and most often resort to code duplication, complex build and validation environments and rigid distribution channels, hurting productivity and market opportunities.

Since technological and economical reasons have put a stop to the increase of clock frequency, performance improvements now come in the form of additional parallelism. It is expected that the number of available cores will grow to reach hundreds or even thousands [1, 11, 16] by 2020.

Applications have a much longer lifetime than hardware. Most of them have been written with a sequential programming model, or at best for a limited amount of parallelism. While the number of cores on a system is increasing by orders of magnitude, it is inconceivable to rewrite millions of lines of code for each new system generation. It will become increasingly important to be able to run an application on an architecture radically different from its original execution target. At run time, different instruction sets, different microarchitectures, a larger number of cores, and different interconnection networks will be encountered.

Of course, it is easy to agree on the need for change in embedded application development. However, while achieving portable functionality is already difficult, achieving both functionality and performance is a major challenge, so far unmatched on heterogeneous hardware.

Interestingly, Graphical Processing Units (GPUs) flourish in an ecosystem dominated by virtualization. This trend is driven by hardware vendors who cannot afford burden of binary compatibility (among other motivations), and by 3D graphics APIs like OpenGL and DirectX. The recent emergence of fully programmable devices boosted processor virtualization for GPUs to another level, with NVidia's PTX intermediate language for CUDA and Apple's usage of LLVM for online partial evaluation of OpenGL stacks [30].

Based on these successes, we propose to extend the application domain of *processor virtualization* (a.k.a. process virtualization) and to combine it with *split compilation*, a flavor of *just-in-time* or *deferred compilation* where optimizations are split over multiple *coordinated steps*. We aim for *performance portability over heterogeneous multiprocessors*. Incremental benefits will be gathered along the way:

- to reduce greatly the burden on maintaining numerous compilers and tools needed to support various plat-

forms (different models of cell phones, game consoles, set-top boxes, smart sensors, wearable devices, etc.); and to provide software developers with a simplified homogeneous view of the target systems;

- to open embedded systems to third-party developers, enabling independent high-performance applications to run not only on the host processor, but also on the more powerful on-chip accelerators;
- to streamline the deployment of applications in enterprise networks or in cloud-computing environments, where compute nodes may be very different from each other; this may include the transparent migration of computations from mobile devices to the cloud;
- to improve the applicability of processor virtualization and the performance of programs running on top of such a layer, leveraging high-level semantics from the C or C++ languages;
- to enable aggressive, target- and context-specific runtime optimizations, even for embedded systems, thanks to split compilation;
- to improve application sustainability, taking advantage of virtualization to let applications survive architectures and to exploit new hardware features or additional degrees of parallelism; and at the same time, to free hardware manufacturers from the constraints of binary compatibility and legacy code.

Our research is implemented in the open source compiler GCC [20] and in the Mono [36] virtual machine, and contributed back to the community [14].

Section 2 reviews the state of the art in processor virtualization and split compilation. Section 3 then details our proposal, and Section 4 illustrates it with examples and interesting directions. We conclude in Section 5.

2. STATE OF THE ART

Let us position our proposal w.r.t. closely related work.

2.1 Virtualization

Processor virtualization was made popular by Java [33] in the late 1990s, and today dedicated versions (Java Micro Edition) specifically target the embedded domain.

CLI (Common Language Infrastructure) is a widespread processor independent format. Initially introduced by Microsoft under the name .NET, it is now an international standard [19, 26]. CLI is multi language and supports *managed* as well as *unmanaged* code. In other words, it can be used for high-level programming paradigms (object orientation, garbage collection...) but it can also express the typical low-level programming style of the C language and it can achieve better performance than Java. There are open source initiatives: Mono [36], Portable.NET [43] or ILDJIT [8], as well as proprietary offers such as Microsoft's MicroFramework [34] for the embedded world.

Even though processor virtualization technology is quite mature and popular, Java and CLI bytecode runs today on the host processor only. In addition, Java applications have the reputation to be slow, probably because the first virtual machines only had an interpreter, or a simple JIT compiler. As a consequence, bytecode formats are a priori perceived as inappropriate for performance intensive applications and for embedded systems. Nevertheless, it has been demonstrated that CLI makes a compact program representation for embedded and general-purpose targets [15], and that the bytecode can be efficiently compiled to native code [13].

2.2 Split compilation

Deferred compilation refers to a compilation process that is decomposed into several steps along the "lifetime" of the program. Traditional bytecode language tool chains distribute the roles among offline and online compilers. Verification and code compaction are typically assigned to offline compilation, while target-specific optimizations are performed by online compilation.

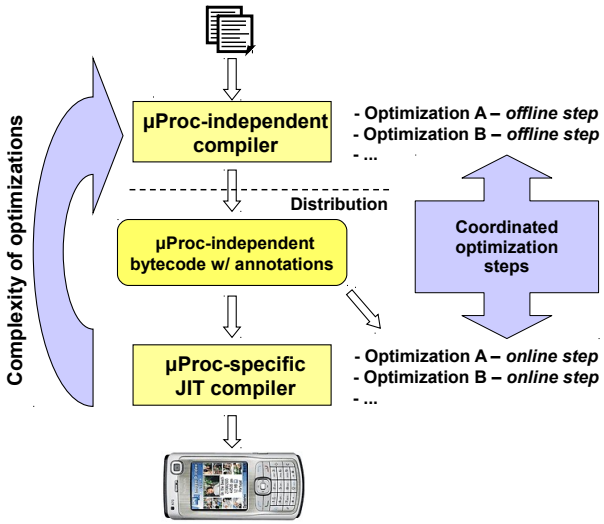

Figure 1: Split compilation flow

Split compilation reconsiders this notion: it allows *a single optimization algorithm to be split into multiple compilation steps*, transferring the semantic information between different moments of the lifetime of a program through carefully designed (bytecode) language annotations. As depicted in Figure 1, a split compilation process may run expensive analyses offline to prune the optimization space, deferring a more educated optimization decision to the online step, when the precise execution context is known. Many JIT compilation efforts tried to leverage the accurate information obtained through dynamic analysis to outperform native compilers: dynamic analysis of type information is a successful example for scripting languages [22], as well as partial evaluation for imperative languages [12]. Split compilation is a concrete path to get the best of both worlds.

This definition of split compilation easily extends to all step of a program lifetime: offline/ahead-of-time compilation, linking, installation, loading, online/run time, and even the idle time between different runs. Each step brings additional knowledge about the run-time environment (shared libraries, operating system, processor, input values), opening the door to new classes of optimizations. Link-time optimizers have the visibility of the whole application and can apply interprocedural analyses and transformations, e.g. alto [35],

or Diablo [44]. Run-time optimizations are applied by JIT compilers [37] and dynamic binary rewriters [3, 17]. Idle time optimizers re-optimize an existing binary, taking into account profiling information produced by previous runs of the application, for example Morph [45].

During the transformation from high-level language, optimizers gain increasing knowledge about the final run-time environment, but they also gradually lose semantic information: arrays become pointers, typed integers become 32-bit amorphous values, etc. There have been many attempts to help compilers with information it cannot derive by itself, including pragmas like OpenMP, or explicit multistage programming like DyC [24] or aC# [10]. On the other hand, LLVM [31] is a widely used framework which demonstrated that lifelong program analysis and transformation can be made available to arbitrary software, and in a manner that is transparent to programmers. For deferred compilation to be effective, high-level information must be propagated while lowering the program representation: it can take the form of annotations in Java class file for register allocation [2, 18, 27], array bound checks removal [38] or side effect analysis [29, 32]. Split compilation generalizes these approaches: it uses annotations and coding conventions in the intermediate language to coordinate the optimization process over the entire lifetime of the program.

3. PROPOSAL

Because hardware's lifecycle is much shorter than software's, hardware vendors generally have to make newer versions of their processors backward compatible, at a high cost. Virtualization lets them introduce radically new and efficient architectures, without having to worry about legacy code. But virtualization can even go beyond mere compatibility. Traditional backward compatibility is limited to functionality; old code can only take advantage of increased clock frequency (which, incidentally, has stopped) and improved microarchitecture, but not of additional hardware acceleration features or increased parallelism.

When a new architecture introduces floating point support that was lacking in the previous release, old code still runs, but it does not take advantage of the newly available hardware. If the application was shipped in a platform-neutral bytecode, the JIT compiler for an ARM Cortex processor could decide to use the most advanced version of the Neon vector extensions, if available. But the JIT compiler for an IBM Cell processor could process the same code and decide to offload some of the numerical computations to a vector accelerator (SPU), running the control-oriented code on the PowerPC core.

Performance critical applications, like games, ship in many versions compiled to cover most of the configurations of the end user: type of graphics cards, version of processor. It is expected that the diversity of systems will increase. Shipping many versions of an application just does not scale, and leads to frustrating experiences for third-party developers and consumers. Future systems will depend on some form of virtualization to ensure portability. But *new techniques need to be developed to ensure performance portability.* The above-mentioned vector extension and heterogeneous parallelization scenarios will push for a dramatic increase in aggressiveness of JIT compilers, and possibly, in changes in bytecode languages to carry the necessary semantical information enabling such complex transformations [4].

Virtualization also alleviates industrial concerns.

Development tools. At present, developers need a complete toolchain for each kind of core present on their systems. Each compiler might (and often does) come from a different vendor, and thus it might be based on its own technology, controlled by its own set of command-line arguments, and expose target-specific behavior to the programmer. Maintaining these tools is a significant burden (and cost) to the developers. Virtualization can offer a single environment to the developers, postponing and concentrating the specialization of the tools closer to the deployed systems. Interestingly, "closer" may not necessarily mean "on", and deferring the specialization may not require JIT compilation technology and overhead: load-time or configuration-time scenarios are possible, especially for deeply embedded or tightly constrained systems.

Debuggability, reproducibility. Because the same application needs to run on different pieces of hardware, current source code contains many conditional preprocessing directives (`#ifdef` in C), and programmers rely on compiler intrinsics and ad-hoc command line flags to drive the optimization. This severely impacts code readability and productivity, and the application binary tested and debugged on a workstation is different from the one that eventually runs on the system. These caveats are unavoidable when platform specialization must be prepared at the source level; but virtualization makes it possible to run the *same bytecode program* on many hardware variants, including the developer's workstation.

Platform openness. Independent software vendors rarely have access to the toolchains needed to program the most powerful parts of the system, namely the DSPs and hardware accelerators. They are given access only the host processor, typically through a Java virtual machine. Virtualization can make the whole platform programmable, opening opportunities to third-party high-performance applications.

To improve programmer productivity on heterogeneous multicore systems, we propose to leverage processor virtualization, combined with an extensive usage of split compilation. As of today, processor virtualization is applied only to the host processor. In order to develop the above mentioned directions, we propose to extend it along three main directions:

- for languages such as C and C++;
- for high performance, competing with offline, native compilation;
- for whole-system programming, including DSPs, accelerators, or grids of computing nodes.

Processor virtualization can be of a great help to program heterogeneous multicore systems. Since the final code generation occurs at run time, mapping and scheduling of computations can be performed across all available processing nodes, independently from their underlying architectures.

Processor virtualization provides the framework for split compilation. The application is compiled in two steps: first,

from source code to the intermediate format; then, from the intermediate format to native code. The former occurs on the developer's powerful workstation, while the latter typically occurs on the system, with full knowledge of the target hardware, but it is CPU and memory bound (other intermediate scenarios are possible).

In the split compilation approach, dynamic optimization is not a replacement for offline optimization: it is a complementary optimization opportunity that leverages (1) any semantical properties distilled by the offline step and carried through the bytecode format, and (2) a wealth of context information not available until run time.

We propose to take advantage of this two-step situation to transfer the complexity of compiler optimizations as much as possible to the first step. Unlike traditional deferred compilation schemes, we do not accept to drop an optimization because it is target-dependent or it may increase code size too much, and because it is too costly to be applied at run time. An expensive analysis and preconditioning of the optimization can be performed and its results encoded into annotations embedded in the intermediate format. The second step can rely on the annotations and skip expensive analysis to implement straightforward code transformations. Annotations may also express the hardware requirements or characteristics of a code module (I/O required, benefits from hardware floating point or vector processing support, etc.).

4. EXAMPLES AND DIRECTIONS

Many complex optimizations, especially those considered too expensive to fit a JIT compiler, can be revisited in the light of split compilation.

Automatic vectorization is currently one of these. The complexity of the transformation is illustrated by the size of the implementation in the GCC compiler: 20k lines, not counting the construction of the SSA form or the induction variable and data dependence analysis. Instead, we showed that a static compiler can produce vectorized bytecode that runs unmodified on many machines, with no or little penalty in the absence of SIMD instructions on the target machine. Table 1 shows the run times of classical kernels, vectorized in their bytecode representation through a set of portable builtins. The JIT compiler on x86 recognizes the vectorization builtins, implements SIMD code generation and shows significant speedups, while the JIT on UltraSparc and PowerPC simply ignores the vectorization and produces code with performance slightly worse to better than scalar (it can be better because the scalarization involves some unrolling of tiny loops). More details can be found in [42].

Split compilation can also enhance classical optimizations, to speed them up [29] or improve their effectiveness [18]. Diouf et al. [18] revisit register allocation, splitting the optimization into coordinated allocation and assignment heuristics. This split leverages fundamental advances in register allocation [7]. Compact, portable annotations drive a linear-time online algorithm, generating code of comparable quality with an optimal offline allocation, and saving up to 40% of the spills on standard Java benchmarks.

Eventually, we believe our approach will help the adoption of the most sophisticated compilation and target-adaptation techniques for embedded development. We are particularly interested in three classes of techniques, which recently matured as research or highly specialized tools, but did not yet integrate traditional development tools:

- Iterative compilation avoids the intrinsic limitations of profitability models to exhibit hard-to-find optimization opportunities. It is very successful at sorting out the interplay of a collection of optimization passes addressing different microarchitectural components [5]. Recent advances improved the practicality of iterative optimization [21], suggesting that virtual machine monitors may be the ideal engines to drive adaptive tuning.
- Whole-program and link-time optimization [31, 44] are well known but little used in production. Again, virtualization can hide the complex deployment of whole-program optimization toolchains, while link-time optimizers can benefit from the higher level semantical information captured in bytecode languages, further enhanced with annotations from split compilation.
- Loop nest parallelization and transformation in the polyhedral model [6, 23] and domain-specific program generation [41] can bring orders of magnitude performance improvements over a generic, portable source code. But they may depend on static properties obtained through whole-program analysis only (precise pointer aliasing information), and they may involve iterative, feedback-directed search. Furthermore, these algorithms require computing resources that are prohibitive in JIT compilers: each hot function may compile for several seconds, using hundreds of megabytes of memory. To make it worse, effective loop nest optimizations are built of long, target-sensitive and input-sensitive transformation sequences [23, 41].

Split compilation appears like the only path to reconcile such radically opposed facts. And virtualization technology naturally feeds the online optimization step with topical context information, while isolating the developer from the complexity of the tool flow. Major research advances will be necessary to follow this path, and to build an effective system achieving portable performance on heterogeneous multicore processors.

We believe that the adaptation of parallel programs in a split compiler will eventually drive the design of new intermediate, bytecode languages. These languages will capture portable, deterministic and composable concurrency information. These properties lead to Kahn process networks [28] as their semantical basis, leveraging the compilation of data-flow languages [9].

To be complete, we should also consider the case of computational kernels where no existing compilation technology rivals manual, target-specific and application-specific optimization. Although not ideal, our approach is still compatible with the occasional reliance on an offline compiler and assembly programming, using native interfaces like `pinvoke` for CLI. A generic version may be provided along with a few native ones for portability, deferring the specialization to the virtualization layer.

Our work is primarily driven by imperative languages like C and C++. But it may incidentally improve the applicability of managed, "productivity" languages like Java, C# and Matlab, as well as functional and even scripting languages. We also consider the raising popularity of control languages like Simulink, and not only as modeling tools but also as code generators. There is an enormous potential for parallelizing and optimizing Simulink compilers in the embedded

	x86 (10^6 iterations)			UltraSparc (10^5 iterations)			PowerPC (10^5 iterations)		
benchmark	scalar	vect.	relative	scalar	vect.	relative	scalar	vect.	relative
vecadd_fp	1197	537	2.2	2810	1947	1.4	999	886	1.1
saxpy_fp	1544	724	2.1	3812	3239	1.2	1460	1101	1.3
dscal_fp	1045	657	1.6	2608	1787	1.5	721	653	1.1
max_u8	3541	227	15.6	3032	3188	0.95	3011	2209	1.4
sum_u8	6707	1277	5.3	8019	8559	0.94	9933	6817	1.5
sum_u16	6710	2547	2.6	8788	11256	0.78	9941	6671	1.5

Table 1: Run times and speedup of split automatic vectorization

system design area [25]. Processor virtualization may alleviate performance caveats of control languages, relying on whole-program optimization and partial evaluation [39, 40].

5. CONCLUSION

Additional performance now come from an increased number of cores and from hardware specialization. This phenomenon dramatically changes the way applications must be handled. Applications will have to exploit an increasing degree of parallelism made available in many different ways: DSPs, vector accelerators, grids of computing nodes, etc.

Processor virtualization is a natural way to address heterogeneity. However, just-in-time (JIT) compilers are constrained by their allocated memory and CPU time budget. In this paper, we propose to combine processor virtualization with split compilation techniques to overcome these limitations. An offline compiler can afford very aggressive analyses to collect relevant information about the application and about the expected benefit of potential optimizations. It may also prepare the adaptation and optimization of a program, building search spaces and predictive models and embedding this semantical information into a generic bytecode format. The JIT compiler can rely on the precomputed information and combine it with up-to-date run-time knowledge to apply the most effective transformations.

As programmers will rely on generic, target-independent parallel programming models, the ability to extract, adapt and map parallel computations to heterogeneous computing resources will be of utmost importance. Yet legacy code will continue to play a major role in shaping the performance of real-world applications. In both cases, combining processor virtualization with split compilation builds a path to efficiently handle the complexity and the diversity of computing resources in the near future.

6. REFERENCES

[1] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Dept., University of California at Berkeley, Dec. 2006.

[2] A. Azevedo, A. Nicolau, and J. Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *Java Grande*, pages 142–151, 1999.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[4] R. L. Bocchino, Jr. and V. S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *VEE'06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 46–56, New York, NY, USA, 2006. ACM.

[5] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation, in Conjunction with PACT'98*, Paris, France, Oct. 1998.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'08)*, Tucson, AZ, USA, June 2008.

[7] F. Bouchez, A. Darte, and F. Rastello. On the complexity of spill everywhere under SSA form. In *LCTES'07*, pages 103–112, 2007.

[8] S. Campanoni, G. Agosta, and S. Crespi Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43(4):11–20, 2008.

[9] D. Cann. Retire Fortran?: a debate rekindled. *Comm. of the ACM*, 35(8), 1992.

[10] W. Cazzola, A. Cisternino, and D. Colombo. [a]C#: C# with a Customizable Code Annotation Mechanism. In *Proc. of the 10th Symp. on Applied Computing*, pages 1274–1278, Santa Fe, NM, 2005.

[11] Computing Systems Consultation Meeting. *Research Challenges for Computing Systems – ICT Workprogramme 2009–2010*. European Commission – Information Society and Media, Braga, Portugal, Nov. 2007.

[12] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Comput. Surv.*, 1998.

[13] M. Cornero, R. Costa, R. Fernández Pascual, A. C. Ornstein, and E. Rohou. An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems. In *HiPEAC'08*, volume 4917 of *Lecture Notes in Computer Science*, pages 130–144, Göteborg, Sweden, Jan. 2008.

[14] R. Costa, A. Ornstein, and E. Rohou. CLI Back-End in GCC. In *GCC Developers' Summit*, pages 111–116, Ottawa, Canada, July 2007.

[15] R. Costa and E. Rohou. Comparing the Size of .NET Applications with Native Code. In *Proc. of the 3rd International Conference on Hardware/Software Codesign and System Synthesis*, pages 99–104, Jersey City, NJ, USA, 2005.

[16] K. De Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'Boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenström, and O. Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050/2007 of *Lecture Notes in Computing Science*, pages 5–29. Springer-Verlag, 2007.

[17] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher. DELI: a new run-time control point. In *Proc. of 35th Annual International Symposium on Microarchitecture*, pages 257–268, Istanbul, Turkey, Nov. 2002.

[18] B. Diouf, J. Cavazos, A. Cohen, and F. Rastello. Split register allocation: Linear complexity without the performance penalty. In *HiPEAC'10*, LNCS, Pisa, Italy, Jan. 2010. Springer-Verlag.

[19] ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.

[20] Free Software Foundation. The GNU Compiler Collection. `http://gcc.gnu.org`.

[21] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Trans. on High Performance Embedded Architectures and Compilers*, 1(1):13–31, Jan. 2007.

[22] A. Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *PLDI'09*, Dublin, Ireland, June 2009.

[23] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3):261–317, June 2006. Special issue on Microgrids.

[24] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report UW-CSE-97-03-03, Univ. of Washington, 1999.

[25] S.-I. Han, S.-I. Chae, L. Brisolara, L. Carro, R. Reis, X. Guérin, and A. A. Jerraya. Memory-efficient multithreaded code generation from simulink for heterogeneous mpsoc. *J. on Design Automation for Embedded Systems, Springer-Verlag*, 2007.

[26] International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 23271:2006 – Common Language Infrastructure (CLI), Partitions I to VI*, 2nd edition, 2006.

[27] J. Jones. *Annotating Mobile Code for Performance*. PhD thesis, University of Illinois at Urbana Champaign, 2002.

[28] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug. 1974. North Holland, Amsterdam.

[29] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 156–167, Snowbird, UT, USA, June 2001.

[30] C. Lattner. The LLVM compiler system. In *Bossa Conference on Open Source, Mobile Internet and Multimedia*, Recife, Brazil, Mar. 2007.

[31] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the International Symposium on Code Generation and Optimization*, Palo Alto, CA, USA, Mar. 2004.

[32] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. Technical Report 2004-5, McGill University - School of Computer Science, 2004.

[33] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, Apr. 1999.

[34] Microsoft. Introducing the .NET Micro Framework. Product Positioning and Technology White Paper, Sept. 2007.

[35] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. alto: a link-time optimizer for the Compaq Alpha. *Software: Practice and Experience*, 31(1):67–101, 2001.

[36] Novell. The Mono Project. `http://www.mono-project.com`.

[37] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} Server Compiler. In *Proc. of the Java Virtual Machine Research and Technology Symposium*, Monterey, CA, USA, Apr. 2001.

[38] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. of the 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 334–354, Genova, Italy, Apr. 2001.

[39] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer-Verlag, 2007.

[40] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In *EMSOFT'09*, pages 215–224, 2009.

[41] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[42] E. Rohou. Portable and efficient auto-vectorized bytecode: a look at the interaction between static and JIT compilers. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*, Jan. 2010.

[43] Southern Storm Software, Pty Ltd. DotGNU project. `http://dotgnu.org`.

[44] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proc. of the International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, Greece, 2005.

[45] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proc. of the 16th Symposium on Operating System Principles*, pages 15–26, Saint-Malo, France, Oct. 1997.