Trimaran: An Infrastructure for Research in Instruction-Level Parallelism

Lakshmi N. Chakrapani[1], John Gyllenhaal[2], Wen-mei W. Hwu[3], Scott A. Mahlke[4], Krishna V. Palem[1], and Rodric M. Rabbah[5]

[1] Georgia Institute of Technology
{nsimhan, palem}@cc.gatech.edu
[2] Lawrence Livermore National Laboratory
gyllen@llnl.gov
[3] University of Illinois, Urbana-Champaign
w-hwu@uiuc.edu
[4] University of Michigan
mahlke@umich.edu
[5] Massachusetts Institute of Technology
rabbah@mit.edu

Abstract. Trimaran is an integrated compilation and performance monitoring infrastructure. The architecture space that Trimaran covers is characterized by HPL-PD, a parameterized processor architecture supporting novel features such as predication, control and data speculation and compiler controlled management of the memory hierarchy. Trimaran also consists of a full suite of analysis and optimization modules, as well as a graph-based intermediate language. Optimizations and analysis modules can be easily added, deleted or bypassed, thus facilitating compiler optimization research. Similarly, computer architecture research can be conducted by varying the HPL-PD machine via the machine description language HMDES. Trimaran also provides a detailed simulation environment and a flexible performance monitoring environment that automatically tracks the machine as it is varied.

1 Introduction

Trimaran is a compiler infrastructure for supporting state of the art research in compiling for Instruction Level Parallel (ILP) architectures. The system is currently oriented toward Explicitly Parallel Instruction Computing (EPIC) [24], and supports a variety of compiler research, including instruction scheduling, register allocation, and software pipelining. The Trimaran compiler infrastructure is comprised of the following components:

- A parameterized ILP architecture called HPL-PD.
- A machine description facility for describing HPL-PD architectures.
- An optimizing compiler with a large suite of optimizations. The compiler is designed such that it may be easily modified and extended by a compiler researcher. The compiler employs an extensible IR (intermediate program

representation) which has both an internal and textual representation, with conversion routines between the two. The IR supports modern compiler techniques by representing control flow, data and control dependence, and many other attributes.
- A detailed HPL-PD architecture simulator which is parameterized via a machine description and provides run-time information on execution time, branch frequencies, and resource utilization. This information can be used for profile-driven optimizations as well as to provide validation of new optimizations.
- A Graphical User Interface (GUI) for configuring and running the Trimaran system.
- Various tools for the graphical visualization of the program intermediate representation and of the performance results.

The infrastructure is used for designing, implementing, and testing new compilation optimizations, as well as the evaluation of various architectural innovations.

Although there are several compiler infrastructures available to the research community, Trimaran is unique in that it is especially geared toward ILP and EPIC research. It provides a rich compilation framework. The parameterized ILP architecture (HPL-PD) space allows the user to experiment with machines that vary considerably in the number and kinds of functional units and register files, as well as their instruction latencies. The modular nature of the compiler and the hierarchical intermediate program representation used throughout the compiler makes the construction and insertion of new compilation modules into the compiler especially easy. The framework is already populated with a large number of existing compilation modules, providing leverage for new compiler research and supporting meaningful experimentation. The Trimaran Graphical Interface makes the configuration and use of the system surprisingly easy. There is a commitment on our part to releasing a robust, tested, and documented software system. Our website (http://www.trimaran.org) provides the latest information on Trimaran, and includes extensive documentation, and facilities to download the infrastructure and other useful material.

The next section briefly describes the Trimaran user community and provides examples of how the infrastructure has been used to further research. In addition, Section 2 also discusses the quality of the code produced by the Trimaran compiler. Section 3 describes the parametric HPL-PD architecture space, and Section 4 presents an overview of the machine description facility for describing HPL-PD architectures. This is followed by an overview of Trimaran compilation technology (Section 5). Section 6 describes the Trimaran simulation environment, and Section 7 concludes the paper.

2 User Community and Field Tests

The Trimaran user community has steadily grown since the first public release of the infrastructure. Today, the user community spans many universities worldwide, and papers which used Trimaran as their evaluation and experimentation vehicle have appeared in numerous conferences on programming languages, compilers, and computer architecture. Trimaran is also used in several classroom settings where it facilitates student projects in compilation technology and VLIW architecture research.

Trimaran has proved to be a versatile infrastructure, and examples of its many uses include research in predicated static single assignment [4] and predicate-aware scheduling [27], software pipelining that is sensitive to register pressure [2], implementation strategies that improve the performance of object oriented codes [3], and optimizations for improving the performance of the memory system [18,19]. Other published works have leveraged Trimaran as a vehicle for architectural design space exploration and processor customization [5,17].

Furthermore, researchers have extended Trimaran so that it can target real architectures such as the Itanium [10], ARM [6], and WIMS (wireless integrated micro systems) [32]. A Trimaran extension which generates ARM assembly is distributed on our website [30]. The Itanium-specific Trimaran is a collaborative effort between the National University of Singapore and the George Washington University [31].

The compiler, in addition to being versatile, is robust and competitive with widely used compilers (e.g., gcc). Trimaran can successfully compile several benchmark suites including SPEC [28], Mediabench [13], and Olden [16] to name a few. Furthermore, an extensive set of results was presented at the 2004 International Symposium on Computer Architecture (ISCA) to demonstrate that Trimaran is on-par with state of the art compilation technology for EPIC architectures [26]. The code quality produced by Trimaran is largely controlled by the compiler front-end which applies a series of transformations as the code is lowered from the input source language. The compilation process begins with application profiling, followed by procedure inlining to reduce function call overhead, alias analysis, classical optimization, and a slew of structural transformation for enhancing ILP (as discussed in the later portions of this paper).

3 Architecture Space

The architecture space targeted by Trimaran is the HPL-PD parametric processor [11]. HPL-PD was designed to promote research in instruction-level parallelism and serves as a vehicle to investigate ILP architectures and the compiler technology needed to effectively exploit such architectures.

HPL-PD is a parametric architecture in that it admits machines of different composition and scale, especially with respect to the amount of parallelism offered. The HPL-PD parameter space includes the number and types of functional units, the composition of the register files, operation latencies and descriptors that specify when operands may be read and written, instruction formats, and resource usage behavior of each operation.

The architecture's instruction set is akin to a RISC load-store architecture, with standard arithmetic and memory operations. It also supports speculative and predicated execution, compiler exposed memory systems, a decoupled branch mechanism, and software pipelining. The following briefly describe the advanced features of HPL-PD which are geared toward enhancing and exploiting ILP. The interested reader can review the HPL-PD architecture specification [11] for a more thorough description of the architecture.

3.1 Speculative Execution

HPL-PD supports control and data speculation. The former represents code motion across conditional branches. When an instructions is *control speculated*, it is moved above a branch and unconditionally executed, whereas previously it was executed conditionally. This transformation is generally safe but may lead to exceptions. For example, if instructions are speculated from the "taken" path following a branch, but the branch resolves to the "not taken" path, then any exceptions that may have been encountered during speculative execution must be ignored. If on the other hand the branch is taken, then the exception must be exposed. HPL-PD provides the necessary support to enable speculative execution [14]. Briefly, if an exception occurs during a speculative operation, the exception is not raised. Instead, a bit is set in the result register to indicate that such a condition occurred. If a non-speculative operation has an operand with its speculative bit set, the exception is immediately raised.

Another form of speculation known as *data speculation* is geared toward increasing the range of code motion for memory instructions [7,23]. For example, a long latency read operation may not be hoisted above an intervening memory write because the load and the store instructions may access (alias) the same location. It is often difficult for the compiler to determine when such a conflict occurs. In order to safely move the data fetch above a store—to better mask an access latency for example—some form of alias detection and recovery support is necessary. The HPL-PD ISA provides the necessary support to enable data speculation.

3.2 Predicated Execution

In HPL-PD, operations can be predicated, or in other words, their execution is guarded by a predicate. For example `(p) LOAD rd = [rs]` is a predicate memory read: the operation is nullified (i.e., the state of the machine does not change) when the predicate is not set (`p` = 0). If on the other hand the predicate is affirmed (`p` = 1) the operation fetches the data stored at address `rs` and writes it to register `rd` (in other words the instruction is issued and allowed to update the state of the machine).

In HPL-PD, a predicate instruction requires an additional operand. The extra operand is a one-bit predicate register whose value guards the execution of the instruction. The value of a predicate register is usually defined using `compare-to-predicate` instructions (`CMPP`) that are a part of the ISA. The ISA provides a rich set of these operations. The `CMPP` instructions are unique in that they define two predicate registers simultaneously, subject to a specified *action*. For example, a `CMPP` may write the value of a comparison to one predicate, and the complementary value to the other predicate. There is a sufficient number of actions defined in the ISA to cover most of the requirements imposed by the various uses of predicates. Furthermore, the architecture permits multiple operations to write into a register simultaneously, provided all producers generate the same value. These write semantics are particularly valuable for the efficient evaluation of boolean reductions as carried out by the `CMPP` instructions.

Predication is most notably used to eliminate hard-to-predict branches. Predicate execution is also used in software pipelining as noted in a subsequent section.

3.3 Exposed Memory Hierarchy

The HPL-PD memory hierarchy is unusual in that it is visible to the compiler. The ISA includes instructions for managing data across the hierarchy, for saving and restoring registers, and for performing run-time data disambiguation.

The instructions that manage the memory hierarchy do so by way of latency specifications and cache directives. For example, a store instructions can specify the highest level in the hierarchy where the stored data should be left for use by subsequent memory operations. A load instructions can also specify a cache directive to indicate its view of the highest level in the memory hierarchy where the fetched data should be left. In addition, a load may specify its expectation as to where its source operand (i.e., data address) is cached. Concomitantly, this hint also specifies the operation latency that is assumed for scheduling.

In order to support run-time disambiguation, HPL-PD provides instructions that can speculatively load data from memory and then verify that the address read was not updated by an intervening store; if it was, then the appropriate register is updated. The ISA also permits a branch to compensation code when the verification fails.

3.4 Branch Architecture

In HPL-PD, there is a rich repertoire of branches, and it includes operations to support software pipelining, and as noted earlier, run-time memory disambiguation. HPL-PD replaces conventional branch operations with two operations. A `prepare-to-branch` instruction loads the target address into a branch target register and initiates a prefetch of the branch target to minimize delays. The instruction may also hint that a branch is taken or not. A `branch` instruction updates the program counter according to its source branch target operand and perform the actual transfer of control.

3.5 Software Pipelining

Software pipelining [12,20] is a technique for exploiting parallelism across iterations of a loop. In software pipelining, the loop iterations are overlapped such that new iterations begin execution before previous iterations are complete. The set of instructions that are in flight at steady state constitute a *kernel*. To reach steady state, a subset of the instruction in the kernel are executed during a *prologue* stage; similarly, another subset is executed during an *epilogue* stage to complete the loop. During the prologue and epilogue stages, predication is used to nullify the appropriate subsets of the kernel.

HPL-PD supports both static and rotating registers. The latter provide automatic register renaming across iterations such that a register `r[i]` in one iteration is referenced as `r[i+1]` in the next.

4 Machine Model

HPL-PD adopts an EPIC philosophy whereby the compiler is responsible for statically orchestrating the execution of a program. Thus, a compiler must have exact information pertaining to the particulars of the architecture definition within the HPL-PD space. In Trimaran, a machine-description (MDES) database specifies those particulars which include the register file structure, the operation repertoire, the set of resources in the architectures (e.g., functional units and memory hierarchy), the resource utilization patterns for each instruction, and the latency descriptors that define when an operand may be read or written after an instruction is issued.

The architecture is defined using a human-readable, high-level machine description (HMDES) language [8]. The description is translated to a low-level language that specifies the same information but in a format that is suitable for a compiler. A MDES Query System (mQS) relays the information to a compiler through a procedural interface. The MDES methodology allows for a retargetable compiler infrastructure and enables experimentation with numerous performance-oriented compiler algorithms (e.g., register allocation, scheduling) as well as architecture-exploration algorithms [29,25] that attempt to discover a machine description best suited for one or more applications of interest.

5 Optimizing Compiler

The Trimaran optimizing compiler is a profile/feedback driven compiler: applications are instrumented and executed using representative workloads to generate information that describes the salient tendencies of the program. For example, profiling information quantifies the likelihood of executing different regions of the program, the predominant control flow paths in the program, and the extent of load-store address aliasing.

The Trimaran compiler generates a control flow graph (CFG) for an input program, where nodes in the graph represent a *basic block* or an atomic unit of execution, and edges connecting nodes represent control flow. The compiler's front-end is charged with applying classic optimizations as well as advanced region formation for boosting ILP. In particular, the compiler leverages profiling information to form traces or long sequences of basic blocks that traverse a frequently occurring control flow path in the CFG. The traces are known as *super blocks* [9] and *hyper blocks* [15]; the later uses predication to merge paths that forge out of a conditional branch. The trace formation algorithms enable many scheduling optimizations and afford more opportunities for control and data speculation.

In addition to these transformations, the compiler's back-end can apply many novel optimizations, many of which are machine specific (e.g., register allocation and scheduling). The back-end is especially rich in scheduling technology and complements the ILP optimizations applied in the front-end.

The Trimaran compiler is highly modularized and is designed to operate in a plug-and-play manner. Thus, the optimizations represent modules that are invoked by a top level driver as dictated by an elaborate set of compiler parameters.

The driver has at its disposal several acyclic scheduling algorithms, including inter-region scheduling [1] where scheduling decisions made in one block affect those made in subsequent blocks. In addition, the compiler can perform modulo scheduling [21], a widely used technique for software pipelining. There are also compiler modules that perform rotating register allocation for software pipelined loops [22], and register allocation for acyclic regions.

The compiler also includes (*i*) techniques to eliminate redundant array loads and stores within loop iterations, and redundant register-to-register moves; (*ii*) if-conversion to form predicated code to enable software pipelining, or to reduce critical path length through a computation; (*iii*) a sophisticated region-based register allocator with predicate code analysis to reduce register saves and restores; and (*iv*) scheduling strategies to tolerate branch and memory latencies.

The intermediate representation (IR) used in the Trimaran compiler is a graph-based IR that is easy to use and extend. The IR is also hierarchical with a *program* node at the root. A program consists of a set of *procedures* which in turn are composed of *blocks*. A block is made up of operations which consist of source and destination operands. The compiler provides many built-in utilities for traversing, transforming, and visualizing the IR. There is also a textual, human-readable equivalent for the internal representation. In Trimaran, all optimizations are IR to IR transformations. This greatly simplifies the design of the compiler and makes the tool chain easy to use and extend.

6 Instruction Set Simulator

The Trimaran infrastructure also includes an instruction set simulator (ISS). The ISS consumes the output of the Trimaran compiler to generate an executable binary which can simulate the original program. The simulator was specifically engineered to allow for an intermixing of HPL-PD and native code. This allows subsets of a large application to be compiled using Trimaran and linked against the remaining portions of the program that are compiled using a native compiler (e.g., gcc). The advantage of such a design is faster simulation.

The main simulation loop processes a table of operations and for each operation it invokes a function that implements the semantics of the opcode. These functions are automatically generated from the machine description file which defines the ISA. Thus, the main simulation core is small and extensible.

The ISS also includes a device stack model to support various components of the architecture such as a branch predictor, an instruction buffer, and any reasonable memory hierarchy configuration. The simulator also tracks many different events and generates a plethora of data and statistics with varying levels of granularity. For example, the ISS can track the execution frequency of a single instruction, as well as its memory system behavior (e.g., how often fetching the instruction resulted in an instruction cache miss). The statistics are also aggregated at the block, procedure, and program levels.

The ISS supports all of the HPL-PD features, including speculation, predication, software pipelining and rotating registers, as well as various register write semantics. It can simulate unscheduled code (i.e., serial execution), unregister allocated code, and allows for intermixing of the different modes. It is also amenable to integration with other tools such as Wattch[1] which would enable power-based experimentation and evaluation.

7 Concluding Remarks

We have described Trimaran, a compilation and simulation infrastructure that was designed to support research in instruction level parallelism. Trimaran is founded upon HPL-PD, a parametric architecture that scales in the amount of parallelism it affords. The architecture and compiler support a variety of ILP-enhancing techniques, including speculation, predication, and software pipelining. Trimaran also provides a detailed instruction set simulator to facilitate experimentation and evaluation of architecture features and compiler optimizations.

We invite researchers who have interests in ILP and EPIC computing to adopt Trimaran and join the user community which currently spans numerous universities worldwide. We encourage users to contribute to the infrastructure to provide a greater repertoire of ideas in a unified environment. This will aid in the comparison of results and the evaluation of ideas. Trimaran is currently evolving to address important research questions that are facing architects and compiler engineers as we forge ahead into a new era of computer system design and organization. We will try to ensure that the evolution takes place in a controlled, coordinated, and timely manner.

Acknowledgments

Trimaran is the result of many man-years of research and development. It began as a collaborative effort between the Compiler and Architecture Research (CAR) Group at Hewlett Packard Laboratories, the IMPACT Group at the University

[1] Wattch is available for download from http://www.eecs.harvard.edu/~dbrooks/wattch-form.html.

of Illinois, and the Center for Research on Embedded Systems and Technology (CREST) at the Georgia Institute of Technology. CREST was the ReaCT-ILP Laboratory at New York University. Many thanks to the former members of the CAR group—especially Santosh Abraham, Sadun Anik, Shail Aditya Gupta, Richard Johnson, Vinod Kathail, Mike Schlansker, Robert Schreiber, Greg Snider, and of course, the late Bob Rau—for their many years of support and contributions, without which, Trimaran would not be possible.

References

1. S. Abraham, V. Kathail, and B. Deitrich. Meld scheduling: A technique for relaxing scheduling constraints. Technical Report HPL-1997-39, Hewlett Packard Laboratories, Feb. 1997.
2. G. Altemose and C. Norris. Register pressure responsive software pipelining. In *Proceedings of the 2001 ACM symposium on Applied computing*, 2001.
3. M. Arnold, M. Hsiao, U. Kremer, and B. G. Ryder. Instruction scheduling in the presence of java's runtime exceptions. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, 2000.
4. L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1999.
5. N. Clark, W. Tang, and S. Mahlke. Automatically generating custom instruction set extensions. In *Proceedings of the 1st Annual Workshop on Application-Specific Processors*, Nov. 2002.
6. S. Furber. ARM *System Architecture*. Addison Wesley, 1996.
7. D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–195, 1994.
8. J. Gyllenhaal, W. Hwu, and B. R. Rau. Hmdes version 2.0 specification. Technical Report IMPACT-96-3, University of Illinois, Urbana, 1996.
9. W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, Jan. 1993.
10. Intel Itanium Processors. `http://www.intel.com/itanium/`.
11. V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-9380 (R.1), Hewlett Packard Laboratories, Feb. 2000.
12. M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.
13. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997. IEEE Computer Society.
14. S. Mahlke, W. Chen, R. Bringmann, R.Hank, W. Hwu, B. R. Rau, and M. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4), Nov. 1993.

15. S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.
16. OLDEN benchmark suite. `http://www.cs.princeton.edu/~mcc/olden.html`.
17. K. Palem, L. Chakrapani, and S. Yalamanchili. A framework for compiler driven design space exploration for embedded system customization. In *Proceedings of the 9th Asian Computing Science Conference*, Dec. 2004.
18. R. Rabbah and K. Palem. Data remapping for design space optimization of embedded memory systems. *To appear in a Special Issue of the ACM Transactions in Embedded Computing Systems*, 2003.
19. R. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.
20. B. R. Rau. Iterative Modulo Scheduling. Technical Report HPL-94-115, Hewlett Packard Company, November 1995.
21. B. R. Rau. Iterative modulo scheduling. Technical Report Technical Report HPL-94-115, Hewlett-Packard Laboratories, Nov. 1995.
22. B. R. Rau, M. Lee, P. Tirumalai, and M. Schlansker. Register allocation for modulo scheduled loops: Strategies, algorithms and heuristics. Technical Report HPL-1992-48, Hewlett Packard Laboratories, May 1992.
23. A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1992.
24. M. Schlansker and B. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, 2000.
25. R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2), June 2002.
26. J. W. Sias, S. zee Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu. Field-testing impact epic research results in itanium 2. In *Proceedings of the 31st annual international symposium on Computer architecture*, June 2004.
27. M. Smelyanskiy, S. Mahlke, E. Davidson, , and H.-H. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, Mar. 2003.
28. STANDARD PERFORMANCE EVALUATION CORPORATION benchmark suite. `http://www.spec.org`.
29. S. Talla. *Adaptive explicitly parallel instruction computing.* PhD thesis, New York University, 2000.
30. TRICEPS: A TRIMARAN-based ARM code generator. `http://www.trimaran.org/triceps.shtml`.
31. TRITANIUM: A TRIMARAN-based Itanium code generator. `http://hydrogen.cs.gwu.edu/tritanium`.
32. Wireless integrated microsystems. `http://www.wimserc.org`.