Leveraging Predicated Execution for Multimedia Processing

Dietmar Ebner Florian Brandner Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstr. 8, A-1040 Wien, Austria
{ebner,brandner,andi}@complang.tuwien.ac.at

Abstract—**Modern compression standards such as H.264, DivX, or VC-1 provide astonishing quality at the costs of steadily increasing processing requirements. Therefore, efficient solutions for mobile multimedia devices have to effectively leverage instruction level parallelism (ILP), which is often achieved by the deployment of EPIC (Explicitly Parallel Instruction Computing) architectures. A characteristical architectural feature to increase the available ILP in the presence of control flow is predicated execution. Compilers targeting those hardware platforms are responsible to carefully convert control flow into conditional/predicated instructions – a process called if-conversion.**

We describe an effective if-conversion algorithm for the CHILI – a novel hardware architecture specifically designed for digital video processing and mobile multimedia consumer electronic. Several architectural characteristics such as the lack of branch prediction units, large delay slots, and the provided predication model are significantly different from previous work, typically aiming mainstream architectures such as Intel Itanium.

The algorithm has been implemented for an optimizing compiler based on LLVM. Experimental results using a cycle accurate simulator for the well known benchmark suite MiBench and several multimedia codecs show a speed improvement of about 18% on average. On the same programs, our compiler achieves a speedup of 21% in comparison to the existing code generator based on gcc.

I. Introduction

With the emergence of modern compression standards such as H.264, DivX, and VC-1, the complexity of multimedia systems ranging from mobile multimedia devices to high definition video systems rises steadily, imposing new demands on both software and hardware design. Highly optimizing compilers are needed to build efficient systems leveraging the particular hardware architecture despite of increasing time-to-market pressure. At the same time, high performance requirements necessitate the ability to execute multiple instructions per cycle. EPIC (Explicitly Parallel Instruction Computing) architectures supporting predicated execution models are more and more applied for multimedia applications.

Compilers targeting those hardware platforms are responsible to explicitly group instructions together and to acquire enough instruction level parallelism to keep the processor busy. The latter is often achieved by eliminating control dependencies sequentializing the surrounding instructions by the use of properly predicated machine instructions whose result is conditionally nullified – a process often referred as *if-conversion*. This allows to execute instructions from different paths in the control flow graph within the same basic block, thereby exposing more parallelism to the scheduler. Additionally, the control flow graph (CFG) is simplified which often enables further optimizations such as software pipelining, vectorization, or certain loop transformations.

An important issue is *when* to carry out if-conversion in the compilation process. One option is to apply it very early during code generation, thereby requiring subsequent analysis and optimizations to deal with predicated code [2]. This allows to take full advantage of predication within the compiler framework. However, it is very hard to estimate the real benefit of a certain transformation at this level, therefore typically requiring a form of reverse if-conversion if it turns out that a transformation has been counterproductive. Another option is to defer if-conversion right before code layout and instruction scheduling [11]. This often facilitates prior passes and allows for more precise cost calculations.

Another questions that has to be addressed carefully is *what* should be converted into a predicated form. In order to achieve efficient execution, a delicate balance has to be found that is strongly dependent on the particular architecture and related scheduling decisions.

In this work, we describe an if-conversion algorithm for the *CHILI* architecture – a novel 4-way VLIW processor by ON DEMAND Microelectronics with extensive support for conditional execution. This platform is specifically designed for digital video processing and mobile multimedia consumer electronic. An overview with a short description of the distinctive predication model is given in Section III.

Our algorithm is implemented as a target architecture dependent component of a static compiler based on LLVM[1] and is applied late in the compilation process right before instruction scheduling/bundling and code layout. It differs from previous work in the special predication model that requires an additional slot per predicated instruction and the inherent support for code duplication; see Section IV. We provide detailed experimental results and conclusions in sections V and VI.

This work is supported in part by ON DEMAND Microelectronics and the Christian Doppler Forschungsgesellschaft.

[1]http://www.llvm.org

II. Related Work

Allen et al. [1] were the first to describe if-conversion – the conversion of control dependencies to data dependencies – in a vectorizing compiler. This transformation enables vectorization of sections of code which otherwise could not be converted by the Parallel Fortran Converter. It is noted that if-conversion has many application areas beyond vectorization.

In [5] Dehnert et al. describe the hardware support of the Cydra 5 for software pipelining. Important are the single-bit Iteration Control Registers (ICRs) which are used to control predicated execution. If-conversion plays an important role in software pipelining for loop bodies up to 20 basic blocks.

Warter et al. [12] were the first to introduce reverse if-conversion that transforms scheduled if-converted code back to the control flow graph representation. They defined a predicate intermediate representation (predicate IR) and transformations from the control flow graph to predicate IR and back. With this transformations, the task of global scheduling is reduced to local scheduling.

Efficient execution of code generated for a processor with predicated execution requires to balance between control flow and predication. August et al. [3] present the partial reverse control framework that allows the compiler to maximize the benefits of predication as a compiler representation while delaying the final balancing of control flow and predication to scheduling time. The partial reverse if-conversion framework achieves great speedups over the hyperblock framework.

Fang et al. [6] describe an algorithm which not only minimizes the number of predicates used for basic blocks, but also moves the predicate assignments as early as possible to relax dependency constraints introduced by the if-conversion. Additionally, common subexpression elimination for if-converted code is presented. No empirical data about the effectiveness of the algorithm is given.

Choi et al. [4] did a comprehensive study to evaluate three different levels of if-conversion aggressiveness measuring the effects on overall execution time, register pressure, code size, and branch behavior. If-conversion could remove 29% of the branch mispredictions, but the speedup is quite small.

Usually, if-conversion is done early in the compilation process. Snavely et al. [11] present a link-time optimizer which does predicate analysis and if-conversion very late in the compilation process at the same time as instruction scheduling and just before code layout. The link-time optimized code is denser and almost as fast as the best code produced by the Intel ecc compiler. For the same programs compiled with the gcc compiler the average speedup is almost 6%.

Hazelwood and Conte [8] developed a lightweight algorithm for if-conversion during dynamic optimization. This algorithm uses dynamic branch prediction information (not including the warm up phase) to apply if-conversion and reverse if-conversion to optimize the code of the static compiler. The method effectively balances the effects of static if-conversion, achieving speedups of up to 14.7%.

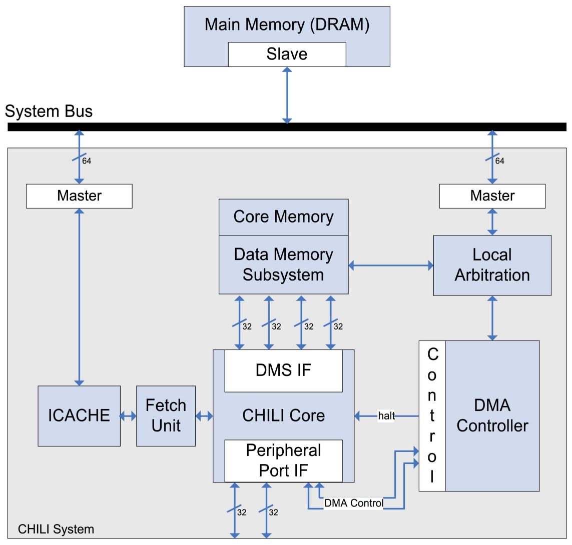

Fig. 1. Overview of the *CHILI* architecture.

III. Target Architecture Description

The effects of if-conversion strongly depend on several architectural characteristics such as the absence or presence of branch prediction, the extend of support for predicated execution, or the number of available branch units, *e.g.*, previous work [10] shows that the performance gain is reduced almost by half when using two instead of one branch unit(s). Therefore, this section describes the most important properties of our target architecture and gives an overview of the provided predication model.

CHILI is a 4-way VLIW (Very Long Instruction Word) architecture specifically aimed for efficient (mobile) video processing. An Overview of the architecture is given in Figure 1. Each slot has access to a general purpose register file offering 64 32-bit registers. Loads and Stores can be issued on each of the four slots and are executed out of order in the data memory subsystem. Branches can only be issued in the first slot and expose a large delay slot of five cycles, *i.e.*, instructions directly succeeding a branch are executed in any case.

Most existing general purpose architectures have either only partial support for predicated execution (mostly restricted to conditional moves, *e.g.*, DEC Alpha, Sun Sparc v9) or nullify the result based on the value of an additional boolean source predicate (Itanium, ARM), which has to be evaluated beforehand. The *CHILI* differs from these architectures in that conditions are evaluated alongside to the instruction to be predicated within the same bundle. Therefore, the full range of binary comparisons is provided in addition to special test instructions that evaluate to true if a particular bit is set or unset respectively. However, these computations require an additional slot in the instruction word. In particular, even slots can be used to evaluate the predicate for the instruction in the directly succeeding slot. The only exception are currently load and store instructions, which cannot be executed conditionally. If multiple instructions are defining a register within the same

```
int min(int a, int b) {
  return a < b ? a : b;
}

min:
{ ret; r0 = r1; if (r2 <= r1) r0 = r2; }
{ nop; nop; nop; nop } //repeated 4 times
```

Fig. 2. Simple minimum computation for the *CHILI* architecture.

bundle, the value produced in the slot with the highest index is kept.

As an example, consider the simple minimum function depicted in Figure 2. Register `r0` denotes the return value while `r1` and `r2` are arguments one and two respectively. Semicolons are used as a delimiter among instructions of the same bundle. Note, that the conditional assignment `if (r2 <= r1) r0 = r2;` occupies two consecutive slots in total. The `ret` instruction has a five cycle delay slot that has to be filled up with no-ops.

The additional costs for predicated instructions in terms of resource usage and code size have to be carefully considered during if-conversion. On the other hand, the large instruction word and the long delay slot of branches usually expose a significant amount of spare resources that can be effectively used for if-conversion. We provide detailed experimental results of our if-conversion algorithm in Section V.

IV. Increasing ILP by If-Conversion

This section describes our if-conversion approach that has been implemented for a `C/C++` compiler backend for the *CHILI* architecture. The compiler is based on LLVM – a carefully designed set of libraries that can be easily combined in order to build optimizing static compilers as well as dynamic code generators. While most parts of LLVM operate on a well defined, target independent *IR* (intermediate representation), our if-conversion procedure operates on an abstract representation of concrete target dependent machine instructions, already after instruction selection and register allocation.

None of those early code generation passes are aware of the VLIW design of the *CHILI* and treat machine instructions as a sequential list of operations. A special scheduling/bundling pass is responsible for grouping them together to VLIW bundles that adhere scheduling and resource constraints and make effective use of branch delay slots.

An important issue is to decide when to carry out if-conversion in the compilation process. Our approach is to perform the transformation right before bundling and code layout. This allows us to treat if-conversion as a separate – optional – phase and keep the rest of the backend small and simple. Previous work [11] shows that such an approach is capable to retain most of the optimization opportunities present in the input program. A major advantage is that code transformations such as spill code insertion have already been carried out. This allows us to if-convert the final machine instructions without taking care of subsequent passes.

It is important to note that the final scheduling and the creation of VLIW bundles is carried out in a dedicated pass after the if-conversion procedure. The main drawback is that we can only roughly estimate the profitability of transformations due to the limited knowledge of the final instruction bundling. On the other hand, this allows the scheduler to effectively exploit the increased parallelism present in the if-converted code in general.

A. Algorithm Description

The overall structure of our if-conversion algorithm is outlined in Figure 3. In each transformation, a single block of the control flow graph (CFG) is merged into one of its predecessors, thereby always preserving the semantics of the original program. Nodes with a single predecessor can be removed entirely after the conversion. A basic block S is said to be *if-convertible* if all instructions can be executed conditionally without side-effects on the remaining program.

A simple example showing the stepwise transformation of a CFG fragment is outlined in Figure 4. The sample assumes that all of the blocks involved are eligible for if-conversion.

Since conditional execution of instructions requires an additional slot, our aim is to execute them unconditionally whenever possible. Therefore, we compute and maintain liveness information for the insertion point (the original location of the branch to be replaced). Instructions without side-effects that do not clobber any register that is live in the predecessor block can be inserted without predication.

Several aspects of the algorithm deserve comment:

- It is important to note that we do not exclude blocks with more than one predecessor from consideration. Thus, those blocks might be effectively duplicated several times. Therefore, it is essential to process the CFG in breath first order to ensure that all predecessors of a particular block have already been considered when visiting a particular node in order to avoid unfavorable code duplication, *e.g.*, compare the transformation sequence depicted in Figure 4 with an order where either block five or six is processed prior to their predecessors.
- Predicates are represented as a triple $(a, \odot, b)$ while a and b are operands and $\odot$ denotes the comparison operator. While a is always a machine register, b might also be an immediate value. Due to the implicit re-evaluation of conditions, we can easily negate conditions at no additional cost by replacing $\odot$ with the inverse operator. However, there are two issues to consider:
  - If there is an instruction that clobbers one of the operands and there is at least one succeeding instruction that has to be predicated, we have to backup the original value using a temporary register and modify the condition accordingly.
  - When if-converting instructions that are already predicated, we have to compute the logical conjunction of both conditions, which is quite costly. Therefore, we maintain a set of available predicates that have already been computed within a block and

```
compute liveness information
for all basics blocks $B$ in breath first order do
    for all successors $S$ of $B$ do
        normalize $S$ and $B$
        if $S$ is if-convertible into $B$ then
            if there is a branch instruction that can be eliminated when if-converting $S$ into $B$ then
                if it is profitable to if-convert $S$ into $B$ then
                    for all instructions $s$ in $S$ do
                        if $s$ has side effects or clobbers any register live at the insertion point then
                            insert an appropriately predicated version of $s$ into $B$
                        else
                            clone $s$ and insert it into $B$
                    update liveness information appropriately
                    if $B$ was the only predecessor of $S$ then
                        remove $S$ entirely from the control flow graph
        cleanup unnecessary branches and merge blocks if possible
```

Fig. 3. General outline of the if-conversion algorithm.

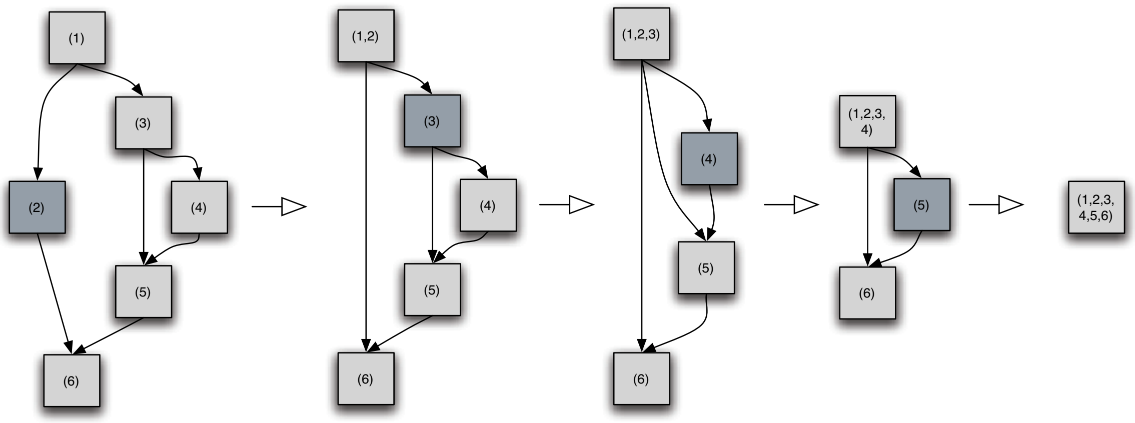

Fig. 4. Stepwise transformation of complex CFG fragments.

have not been invalidated by consecutive instructions. This is a very common case for large blocks, *e.g.*, instructions in basic block four in Figure 4 have to be predicated with the logical conjunction of the conditions corresponding to edges $(1, 3)$ and $(3, 4)$.

- If-conversion is complicated by basic blocks that might fall through into their immediate successor. Therefore, we identify those blocks and *normalize* them, *i.e.*, explicit jump instructions are inserted. After if-conversion, we perform the inverse operation and cleanup those unnecessary branches. Additionally, basic block chains without side exits and entries are merged into a single block. In general, this exposes more parallelism to the basic block local scheduling/bundling pass.

A very delicate question is to determine whether it is *profitable* to if-convert a particular block. Therefore, we use a combination of several heuristic criteria:

1) *Loop Depth*: Let $l(B)$ denote the loop nesting level of block B, then we do not if-convert a block B' into B if $l(B') < l(B)$, *i.e.*, we do not move code into loop bodies.
2) *Block Size*: The sum of latencies of instructions combined with additional costs for instructions that have to be predicated is compared to an experimentally determined, *architecture dependent threshold*; see Section V. The value is intended to be a measure for the size of the block. Instructions with high latencies are accounted accordingly. For blocks with more than one predecessor, a different – much smaller – threshold is applied to account for additional costs in terms of code size.
3) *DDG Depth*: Typically, it is very hard to acquire enough ILP to keep a 4-way machine busy and most instructions that have been if-converted can be efficiently scheduled along with the remaining blocks. However, converting blocks with a lot of dependencies among the instructions can severely increase the execution time of (possibly frequent) paths that are taken if the particular predicate is false. Thus, we additionally compute the *maximum depth* of the corresponding data dependence graph (DDG) and exclude those basic blocks from consideration that exceed a specific threshold.

benchmark	lines	code size		
	of code	no if-conv	full if-conv	
CRC32	136	736770	736502	-0.03%
FFT	276	839581	839401	-0.02%
adpcm	183	695836	694324	-0.21%
basicmath	326	936928	933069	-0.41%
bitcount	549	656739	656172	-0.08%
blowfish	1185	722480	721897	-0.08%
dijkstra	142	980352	980168	-0.01%
h.263	4789	1802461	1757438	-2.49%
jpeg	15026	1929910	1899473	-1.57%
mp3	8758	1193236	1166789	-2.21%
sha	205	699281	698345	-0.13%
stringsearch	3130	651436	651192	-0.03%
susan	1454	1102714	1085149	-1.59%

TABLE I
BENCHMARK CHARACTERISTICS

V. EXPERIMENTAL RESULTS

We evaluate our algorithm using a cycle accurate simulator for the *CHILI* architecture. There is no operating system layer – all benchmarks are simulated to run directly on the hardware. Since simulation is a very time consuming task, the default inputs for some of the benchmarks are slightly shortened.

Most of the test programs are taken from the MiBench suite [9], [7], a free and commercially representative benchmark suite for embedded architectures. We omit those that cannot be compiled with `newlib` because of dependencies on operating system features such as sockets and pipes. In addition, we add some multimedia decoders such as `h.263`, `jpeg`, and `mp3`. Some characteristics of our test programs are shown in Table I.

As we describe in Section IV, the decision if a block is going to be if-converted depends on several architecture dependent thresholds. The average improvement for various combinations of instruction and depth threshold is shown in Figure 5. The results have been gathered with a set of ten benchmarks that could be simulated in reasonable time (less than 30 minutes per instance). For blocks with more than one predecessor, we apply a constant threshold of two. It can be seen that too large thresholds even lead to a decrease of performance. However, this effect is slightly obscured by the missing support for conditional loads and stores, which are likely to be present in large basic blocks.

The speedup achieved by if-conversion for our benchmark set is shown in Figure 6. While some small instances do not offer many possibilities for if-conversion, benchmarks such as `adpcm` are sped up by an astonishing factor of 2.82. The average improvement over the whole benchmark set is 18%. Interestingly, a significant fraction of the speedup is already achieved by simple conditional move instructions (7%). Ignoring the huge speedup of `adpcm`, the average improvement is 5% and 3% respectively.

In contrast to our expectations, if-conversion seems to have a throughout positive effect on code size; see Table I. None of the benchmarks has been increased while the average savings are about 0.71%.

Interestingly, code duplication does not have much effect on both performance and code size. The average improvement in code size is slightly decreased from 0.71% to 0.69% while the speedup in comparison to a version with disabled code duplication is less than one percent.

In comparison to an existing compiler based on `gcc 4.2` that has a much more conservative approach to if-conversion, the average speedup is about 21%; see Figure 7.

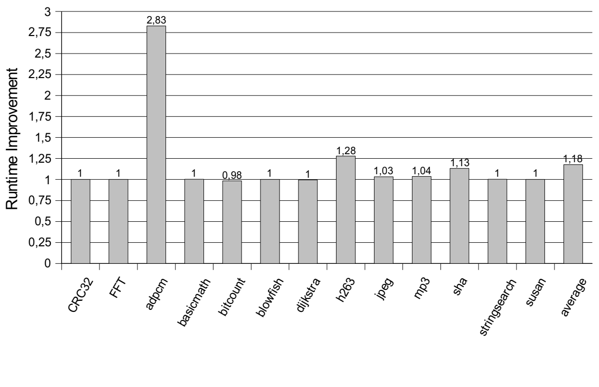

Fig. 6. Speedup achieved through if-conversion for the LLVM compiler.

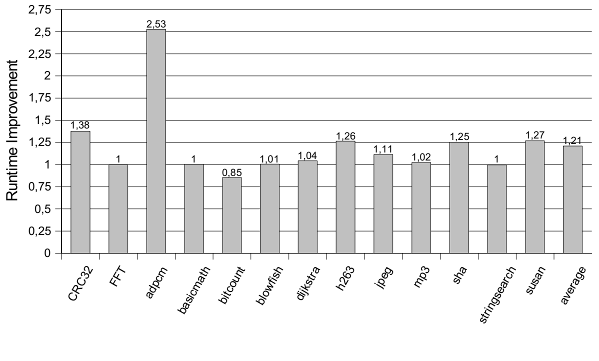

Fig. 7. Speedup in comparison to the existing compiler based on `gcc`.

VI. CONCLUSIONS

In this work, we describe a simple and yet effective procedure to convert control dependencies into data dependencies for an embedded 4-way VLIW multimedia processor with full predication support. Our algorithm has been implemented for a static compiler backend based on the LLVM framework. In contrast to most previous work, if-conversion is done very late in the compilation process. Apart from a subsequent scheduling/bundling pass, the backend is neither aware of predicated execution nor of the VLIW nature of the target platform.

Experimental results using a cycle accurate simulator for the well known benchmark suite MiBench and several multimedia codecs show a speed improvement of about 18% on average. On the same programs, our compiler achieves a speedup of 21% on the best code produced by the existing code generator based on `gcc`. Interestingly, code duplication does neither have a big effect on the resulting code size nor on runtime.

For some of the benchmarks, if-conversion shows no or even slightly negative effects. Apparently, this is due to bad

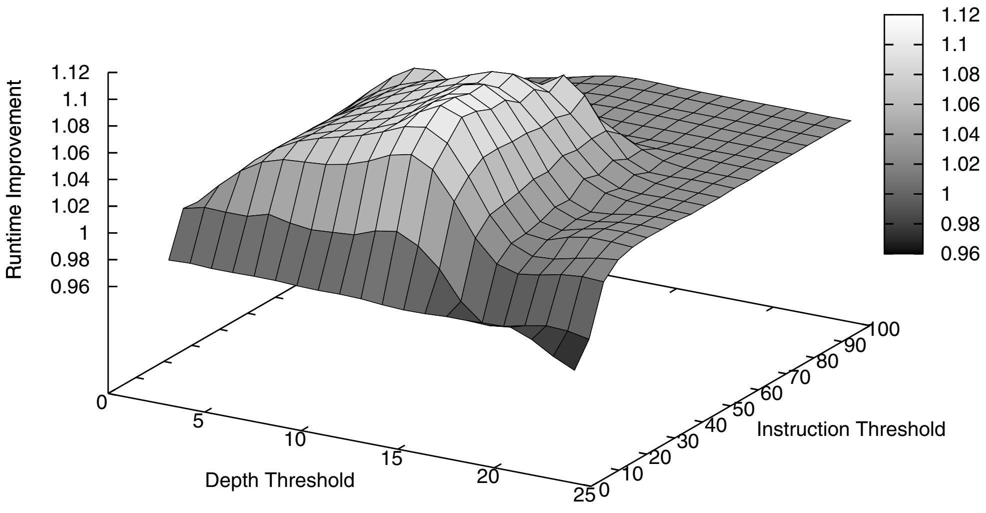

Fig. 5. Experimental verification of architecture dependent thresholds.

if-conversion decisions, which are based on criteria that do not depend on the scheduling of the surrounding blocks. This keeps the compiler backend simple and modular but appears to be a major burden for further improvements. Therefore, future work will include a tighter integration of the if-conversion procedure with the existing scheduling/bundling pass and/or reverse if-conversion techniques.

REFERENCES

[1] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe D. Warren. Conversion of control dependence to data dependence. In Alan Demers, editor, *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Austin, TX, January 1983. ACM SIGACT and SIGPLAN, ACM Press.

[2] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 92–103, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[3] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. The partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming*, 27(5):381–423, 1999.

[4] Youngsoo Choi, Allan D. Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 182–191, Austin, Texas, December 1–5, 2001. ACM/IEEE.

[5] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the cydra 5. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 26–38, New York, NY, USA, 1989. ACM Press.

[6] Jesse Zhixi Fang. Compiler algorithms on if-conversion, speculative predicates assignment and predicated code optimizations. In David C. Sehr, Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing, LCPC'96 (San Jose, California, August 8-10, 1996)*, volume 1239 of *Lecture Notes in Computer Science*, pages 135–153. Springer, 1996.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[8] Kim M. Hazelwood and Thomas M. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, pages 71–80, Philadelphia, October 15–19, 2000. IEEE Computer Society Press.

[9] MiBench Website. http://www.eecs.umich.edu/mibench/.

[10] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 138–150, New York, NY, USA, 1995. ACM Press.

[11] Noah Snavely, Saumya Debray, and Gregory Andrews. Predicate analysis and if-conversion in an itanium link-time optimizer, October 01 2002.

[12] Nancy J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Reverse If-Conversion. *ACM SIGPLAN Notices*, 28(6):290–299, June 1993.