Hardware-Aware Compilation

25

Aviral Shrivastava and Jian Cai

Abstract

Hardware-aware compilers are in high demand for embedded systems with stringent multidimensional design constraints on cost, power, performance, etc. By making use of the microarchitectural information about a processor, a hardware-aware compiler can generate more efficient code than a generic compiler while meeting the design constraints, by exploiting those highly customized microarchitectural features. In this chapter, we introduce two applications of hardware-aware compilers: a hardware-aware compiler can be used as a production compiler and as a tool to efficiently explore the design space of embedded processors. We demonstrate the first application with a compiler that generates efficient code for embedded processors that do not have any branch predictor to reduce branch penalties. To demonstrate the second application, we show how a hardware-aware compiler can be used to explore the Design Space of the bypass designs in the processor. In both the cases, the hardware-aware compiler can generate better code than a hardware-ignorant compiler.

Acronyms

ADL	Architecture Description Language
BRF	Bypass Register File
BTB	Branch Target Buffer
CFG	Control-Flow Graph
CIL	Compiler-In-the-Loop
DSE	Design Space Exploration

A. Shrivastava (✉)
School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
e-mail: aviral.shrivastava@asu.edu

J. Cai
Arizona State University, Tempe, AZ, USA
e-mail: jian.cai@asu.edu; jcai19@asu.edu

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_26

HPC Horizontally Partitioned Cache
ISA Instruction-Set Architecture
MAC Multiply-Accumulator
OT Operation Table
RT Response Time
SPU Synergistic Processor Unit

Contents

25.1 Introduction

Hardware-aware compilation refers to the compilation that exploits the microarchitectural information of the processor to generate better code. Minimally, compilers only require information about the Instruction-Set Architecture (ISA) of the processor to generate code. This ISA-dependent compilation is often good-enough to generate code for high-performance superscalar processors, in which the hardware may drastically modify the instruction stream (e.g., break complex instructions into simpler microinstructions, fuse simple instructions into complex macro-instructions, reorder the instruction execution, and perform speculative and predictive computations) for efficient execution.

However, the processors in embedded systems, or embedded processors, are characterized by lean designs and specialization for the application domain [12, 16]. To meet the strict multidimensional constraints of the embedded systems, customization is very important. For example, even though register renaming improves performance in processors by avoiding false data dependencies, embedded processors may not be able to employ it because of the high power consumption and the increased complexity of the logic. Therefore embedded processors might deploy a "trimmed-down" or "lightweight" version of register renaming, for example, register scoreboarding, which provides a different trade-off in the cost, complexity, power, and performance of the embedded system. In addition, designers often implement some irregular design features, which are not common in general purpose processors, but will lead to significant improvements in some design parameters for the relevant set of applications. For example, several cryptography application processors come with hardware accelerators that implement the complex cryptography algorithm in the hardware. By doing so, the cryptography applications can be made faster, and consume less power, but may not have any noticeable impact on normal applications. Embedded processor architectures often have such application-specific "idiosyncratic" architectural features. And last but not the least, some design features that are present in the general-purpose processors may be

entirely missing in embedded processors. For example, support for prefetching is now a standard feature in general-purpose processors, but it may consume too much energy and require too much extra hardware to be appropriate in an embedded processor.

How can we effectively compile for such uniquely designed embedded processors? Just the information about the ISA is not enough. A good uniquely designed compiler needs microarchitectural information, including sizes of caches, buffers, and execution policies (e.g., register scoreboarding, branch prediction mechanism, etc). Many of these microarchitectural features are independent of the ISA but affect the performance very significantly [20, 28]. By knowing about these microarchitectural features, compilers can design a plan for efficient execution. For example, popular compilers such as GCC [13] and Clang [19], inlines many functions and unrolls loops to improve the run-time performance of applications. However, these optimization techniques increase program code size and thus may not be usable for embedded systems that have very limited instruction memory. To accommodate such diversified and sometimes multidimensional design restraints, not only the compiler must be aware of the memory size of the processor but also make sure that the compiled code can reside in the available memory. A compiler that uses microarchitectural information to generate efficient code is called a hardware-aware compiler.

Figure 25.1 shows the general flow of a hardware-aware compiler. The architectural description is provided along the input program to the compiler, in an Architecture Description Language (ADL). An ADL is a formal language that is used to describe the architecture of a system, including the memory hierarchy, pipeline stages, etc. Examples of ADLs include EXPRESSION [11], LISA [34], and RADL [30]. By taking into consideration the microarchitectural features described by the ADL, the compiler can generate a code that is better optimized for the target architecture. For example, by taking into consideration the memory access timing, the compiler may be able to generate better schedules of instructions for execution [8, 9].

Clearly a hardware-aware compiler is valuable as a production compiler, where it is used to generate carefully tuned code for the target microarchitecture, but it is

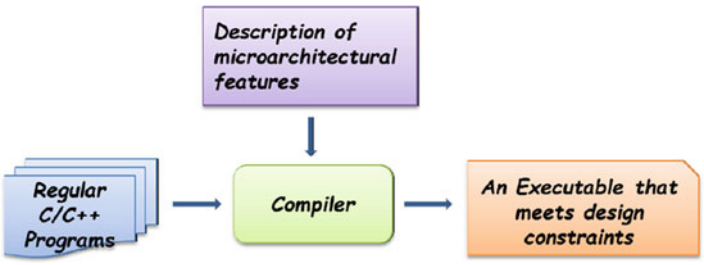


Fig. 25.1 Hardware-aware compiler uses the microarchitecture description of the processor to generate efficient code

also (and arguably even more) valuable for the design of the embedded processor itself. The typical way to design embedded processors, i.e., to determine the microarchitectural configuration – which microarchitectural features to keep in the processor, which execution policy, what buffer, which cache sizes, etc. – is through a simulator-based DSE. In this methodology, a cycle-accurate simulator of the processor with different microarchitectural features is designed. The applications are executed on the simulator to figure out which microarchitectural configuration works best. This methodology relies solely on processor simulators for Design Space Exploration (DSE) on traditional superscalar processors, since the code quality is not very important in them. However, including compiler in the DSE loop is more important for efficient designs of embedded processors, where a compiler can have very significant impact on the eventual power and performance of the application. In the Compiler-In-the-Loop (CIL) DSE methodology, the compiler is used to generate a code for each microarchitectural variation, and the best design is selected. This methodology enables us to pick microarchitectural configurations that may not be as effective by themselves, but lend themselves to very effective use by the compiler, and achieve superior power saving and performance improvement. Such configurations will be disregarded by the traditional simulation-only DSE.

In summary, hardware-aware compilers that use the microarchitectural information about the processor to generate better code can improve the power consumption and performance of embedded processors. They can be used both as a production compiler and be used in the compiler-based processor design. We dive into both of these uses of the hardware-aware compiler in the rest of this chapter. In the next section, we describe the use of a hardware-aware compiler as a production compiler. We present an example of a compiler that generates good code for embedded processors that do not have any branch predictor at all. Branch predictors, though very useful in eliminating most of the branch penalty, are costly (require a lot of hardware) and drain significant amount of power. As a result, some embedded processors may choose to drop them. Note that the presence/absence of a branch predictor does not affect the ISA but has a very significant impact on the power consumption and performance of the execution. Indeed without a branch predictor, all branches will incur branch penalty, and this will be excessive for execution. However, a compiler can help. Instead of expensive branch predictors, embedded processors may choose to have a branch hint instruction, which can indicate to the processor the direction of the imminent branch. If the application developer or the compiler can insert these branch hint instructions at the right places in the code, it can ameliorate most of the branch penalty and result in efficient execution.

Next we will describe how a hardware-aware compiler can be used as an effective tool in the design of embedded processors. We explain this through the example of designing the bypasses in the processor. In pipelined processors, even though the result is evaluated, it cannot be read by the next instruction (if there are no bypasses), until the instruction is committed, and writes its results in the register file. This pipeline penalty due to data dependencies among the instructions can be alleviated to a large extent by using bypasses that forward the results of instructions after evaluation to the operand read stage of dependent instructions. However, processors

now feature extremely long pipelines, often more than 20 stages [3, 6], and a full bypassing, e.g., bypasses from all the later stages of the pipeline to the operand read stage, can be extremely complex and expensive. Skipping some bypasses can reduce the overhead of bypassing logic. However, which bypasses to remove? Clearly, the bypasses that are used least often can be removed, but the compiler has an important role to play in this. If the compiler can reschedule the instructions around the missing bypasses, then the effect of the missing bypasses can be eliminated. The following section first describes, given a partial bypass configuration (i.e., not all bypasses are present), how do we reschedule the code so as to avoid the missing bypasses and then shows the results of DSE with and without this bypass-sensitive compiler in the loop.

For readers that are interested to learn more about related topics, ▶ Chap. 26, "Memory-Aware Optimization of Embedded Software for Multiple Objectives" introduces compiler-based techniques that map applications to embedded systems with scratchpad memories, focusing on minimizing the worst-case execution time of the applications. ▶ Chapter 27, "Microarchitecture-Level SoC Design" presents typical system-on-chip design flow and detailed issues in power modelings, thermal, and reliability, as well as their relation, and presented some interesting solutions.

25.1.1 Hardware-Aware Compilers as Production Compilers

A hardware-aware compiler can be used as a production compiler to generate a code for embedded systems once the microarchitecture is fixed. Researchers have discovered several use-cases for hardware-aware compilers. Muchnick [23] has developed the concept of Response Time (RT) and RT-based compiler that reschedules instructions to minimize the data dependence penalty in processors. A RT specifies how an operation may use the resources of a processor as the operation executes. Their compiler uses the specification of the pipeline of the processor as an input. Using this, it can create RT for the given instructions and detect conflicts among them – the structural and data hazards – so as to generate better instruction scheduling [21, 31]. Bala and Rubin [1] and Proebsting and Fraser [26] proposed compiler techniques that use the finite-state automaton (FSA), a derivative of the RT, to further speed up the detection of pipeline hazards during instruction scheduling. These approaches improve the power and performance of execution.

Hardware-aware compilers have also been proposed to help hide memory latency [25]. The most important source of memory latency in processors is the cache miss penalty, as cache misses typically take orders of magnitude longer time than cache hits. Grun et al. developed a compiler optimization that uses accurate timing information of both memory operations and the processor pipeline to exploit memory access modes, such as page mode and burst mode, so as to allow the compiler to reorder memory operations to help hide the memory latency [8]. They later extended the work and used memory access timing information to perform aggressive scheduling of memory operations, so that cache miss transfers can be overlapped with the cache hits and CPU operations [9]. For example, an instruction

that will cause a cache miss (known by cache analysis) can be scheduled earlier so that the following cache hits to the same cache line will not be stalled while the cache line is being transferred.

Hardware-aware compilers have also been proposed to reduce the power and temperature. Power gating [17, 24, 27] is one such application used in integrated circuit design to reduce the leakage power of processors. Leakage power already contributes to more than 30% of the power consumed by the processor. But by turning off the unused blocks, leakage power of that block can be reduced. However, power gating will backfire if the power spent in turning off and turning on an execution unit is more than the power saved while it is power gated or if we turn on the block too late, and there is a performance penalty corresponding to it. As a result, prediction-based techniques to power-gated blocks are not as effective. However, a compiler can analyze the application and find out regions of code where a functional block is not going to be used. If the functional block is going to be unused longer than a threshold of time, it can be safely power gated to minimize the leakage power of the functional block. To prevent such undesired leakage, the compiler can be utilized to analyze the control flow graph to predict the idle cycles of the execution units and ensure that power gating is applied only if the power saved during these cycles is greater than the power used to turn on/off the execution unit.

Another example of hardware-aware compilation to reduce power consumption can be found in computer architectures with Horizontally Partitioned Cache (HPC). An HPC architecture maintains multiple caches at the same level of memory hierarchy (in contrast to one cache per level to traditional computer architectures). Thanks to caching different kinds of data in separate caches to avoid interference between each other, e.g., between scalar variables and arrays, the HPC architecture is able to reduce the number of cache misses, which directly translates to the improved performance and abated power consumption. Moreover, HPC architectures include at the same level of memory hierarchies one or more small additional caches, aside the large-sized main cache. For example, in the Intel XScale [15], the L1 caches consist of the 32 KB main cache and a 2 KB additional cache. The additional caches typically consume less power per access, which further decreases the power consumption. Although the benefits of the HPC architecture are inviting, it is nontrivial to exploit such an architecture as its performance is highly dependent on the design parameters. Compiler techniques can be used to explore these parameters and carefully partition data to achieve the maximum benefit. For example, Shrivastava et al. identified the access pattern of data, and cached data with temporal locality in the main cache, while leaving data with spatial locality to the additional caches [29]. This is because the size of a cache does not affect the miss rate of memory accesses to data that exhibits spatial locality, while on the other hand, a larger (main) cache is able to have a higher chance to retain the data that shows temporal locality for repeated accesses.

For the rest of this subsection, we will present and detail a software branch hinting technique for processors without hardware branch prediction, but a simple software branch hinting mechanism [22].

25.1.1.1 The Case for Software Branch Hinting

Control hazards or branching hazards pose a serious limitation on the performance of pipelined processors, which becomes worse as the pipeline depth grows. A branch predictor that predicts the direction (taken or not taken) and the target address if the branch is to be taken can solve this predicament. Branch predictors are typically implemented in hardware so as to handle dynamism of branches. However, branch predictors can be expensive in both the area and power [4, 18]. As multi-core processors become increasingly popular even in embedded systems, some embedded multi-core processor designers remove the hardware branch predictors to meet the power cap while still being able to accommodate more cores. The IBM Cell processor [5], in an effort to improve its power efficiency, removes the hardware branch predictors from its Synergistic Processor Unit (SPU) coprocessors.

Doubtlessly, the lack of branch prediction will cause significant performance penalty. Table 25.1 manifests the huge overhead caused by branches when running some typical embedded benchmarks due to the lack of hardware branch prediction. To prevent such extreme performance loss, processors without hardware branch prediction may provide instructions for software branch prediction or software branch hinting, as the IBM Cell processor does. Branch hint instructions must be used wisely in such processors, in order to achieve comparable or even better performance than hardware branch prediction.

A branch hint instruction typically predicts the target address a branch will jump to when the branch is actually taken. This implies that such an instruction must be inserted only if it is for sure that the branch will be taken, to avoid the misprediction from slowing down the program execution. Fortunately, there have been many research works for predicting the direction of a branch [2, 32, 33] with pretty good accuracy. However, even if we know a branch is taken, finding an appropriate place in the program to insert the branch hint instruction is a nontrivial task. On the one hand, it takes time to set up the branch hint instruction, so the hint must be executed early enough to be recognized by the branch to be hinted. In other words, the branch hint instruction must be inserted early enough before the branch to take effect. On the other hand, there is also the restriction on the number of branch hint instructions that can be activated at the same time. Therefore, simply bringing forward the insertion point of a hint may cause problems in some cases. For example, in the IBM Cell processor, only one active branch hint instruction is allowed. So if there are two branches close to each other in the program, placing both hints of the two branches before the first branch (i.e., the second hint is also placed before the first branch to ensure there is enough time left for the second

Table 25.1 The percentage of execution time spent in branch penalty of typical embedded applications without branch prediction in Cell SPU

Benchmark	Branch penalty (%)
cnt	58.5
insert_sort	31.4
janne_complex	62.7
ns	50.9
select	36.2

branch to be hinted) may cause the effect of the first hint overwritten by the second hint, if the second hint has been activated by the time the first branch is executed. This will cause the misprediction at the first branch, force the execution to stall, and wait for the target address to be recalculated.

While the insertion of branch hint instructions can be done by programmers manually, it can be a tedious and time-consuming process. A compiler-based solution may be preferred. In the rest of this subsection, we will present a compiler-based approach for minimizing the branch penalty in processors with only software branch prediction. We will first introduce the model used for the cost function of branch penalties. It is based on the number of cycles between the hint instruction and the branch instruction, the taken probability of the branch, and the number of times the branch is executed. Subsequently, three basic methods for reducing branch penalties using the branch hint instruction are introduced and detailed:

(i) A no operation (NOP) padding scheme that inserts NOP instructions before a branch to leave enough time interval for its hint to be set up, for small basic blocks without enough margin originally.
(ii) A hint pipelining technique that allows two very close branches where originally only one of them can be hinted, to be now both hinted.
(iii) A loop restructuring technique that changes the loop structure so the compiler can insert the hints for more branches within the loop. The heuristic that combines and applies these basic methods to the code prudently is also briefly explained. Finally, experimental results collected are examined to demonstrate the efficacy of the technique.

The discussion will be based on the SPU coprocessor in the IBM Cell processor. However, the presented technique is applicable to other processors with only software branch prediction. Also, we will assume that every instruction takes one cycle for the sake of simplicity, although this is not necessarily true.

25.1.1.2 Mechanism of Software Branch Hinting

Figure 25.2 shows the overview of how a hint instruction works. The execution of a hint instruction comprises two stages: (i) launching the operation and setting up and (ii) loading the target instruction. Similar to hardware branch predictors, software branch hinting employs a Branch Target Buffer (BTB) to predict the target of a taken branch. When a hint instruction is executed, it needs to search the BTB and see if it can find any matched entry, and update the BTB if it fails to find one. This is done in the first stage. Once this stage is over, the hint instruction will start to fetch the target instruction into the hint target buffer. By default, the next instruction will be loaded to the in-line prefetch buffer, so the processor can fetch the instruction and continue the execution. However, when a branch instruction is identified, its PC address is used to search for any matching BTB entry (the BTB would have been updated, in the presence of a hint). If any entry is found matched, the processor will then instead fetch the next instruction from the hint target buffer.

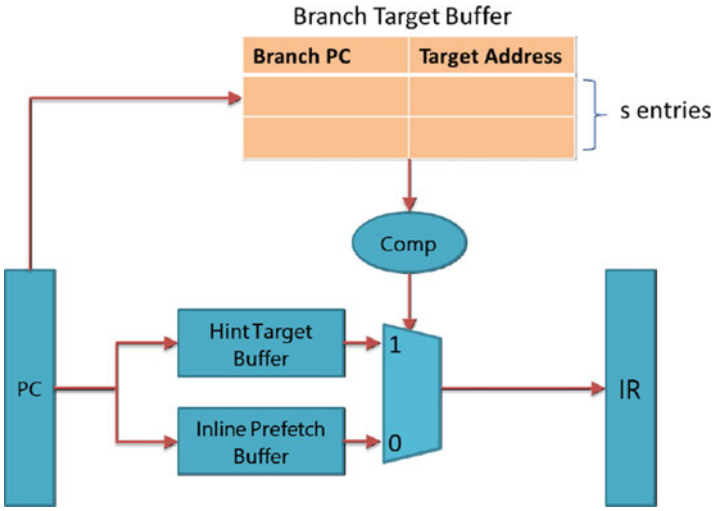


Fig. 25.2 The overview of software branch hint instructions

Given such mechanism of software branch hinting, we should readily identify the three critical design parameters that will seriously affect the performance of resultant implementation:

- d: the number of cycles used for starting up the operation
- f: the number of cycles to load the target instruction
- s: the number of BTB entries

The parameter d decides the minimum interval between a hint and the branch it aims to hint. In other words, a hint instruction must be executed at least d cycles earlier than the branch instruction in the program for it to take effect. After the startup of the hint instruction, a request is made to the arbiter [14] to load the target instruction from main memory into the hint target buffer. This is because in the cell processor, SPUs cannot access the main memory directly, so code and data must be first loaded into the local storage. This stage will take f cycles to complete. Once this stage is finished, the hint instruction is also completed. Therefore, if the hint instruction is executed $d + f$ cycles earlier, the branch it hints can be executed without any stall. In particular, if the branch is actually not taken, it will still wait for the hint instruction to load the incorrect target instruction, and then start over to load the correct instruction.

The number of BTB entries, s, decides size of the hint target buffer, since the buffer must be large enough to hold the target instructions for all the active hint instructions. For example, each SPU of the cell processor has only one entry in the BTB. Therefore, at the same time, only one active hint instruction is allowed for applications run on SPU. The bigger s is, the larger the BTB, and the more power consumption. Therefore, s is usually small.

25.1.1.3 Cost Model of Branch Penalties Under Software Branch Hinting

The branch penalty of a branch with software branch hinting can be modeled as the expected value of the penalty when the branch is successful predicted and when it is mispredicted, respectively. From our previous discussion of the software branch hinting mechanism, we know the branch penalty is related to the number of cycles between a hint and the branch to be hinted, whether the branch is correctly predicted. Therefore, the branch penalty can be modeled as below:

$$Penalty(l, n, p) = Penalty_{correct}(l) \times np + Penalty_{incorrect}(l) \times n(1-p) \quad (25.1)$$

where l, n, and p, respectively, represent the number of cycles between the hint and the branch, the number of times the branch is executed, and the branch probability. We assume n and p are given in our discussion. To find out the relation between the branch penalty and l when a branch is predicted correctly ($Penalty_{correct}(l)$) or incorrectly ($Penalty_{incorrect}(l)$), a synthetic benchmark that includes only a branch hint instruction and the branch instruction to hint is run in the SPU in the IBM cell processor. The hint and the branch are separated by *lnop* instructions (one type of NOP instructions). By increasing the number of *lnop* instructions between the hint and the branch, the change of branch penalties can be inferred through the variation of the execution time.

Two types of NOP instructions are available in the SPU, thanks to its dual-issue nature – *nop* for the *even* pipeline and *lnop* for the *odd* pipeline. The *even* pipeline is used to execute fixed point and floating point arithmetic operations, while the *odd* pipeline is used to execute memory, logic, and flow-control instructions, which include the branch instruction and the branch hint instruction. By filling only *lnop* and branch/hint instructions, the SPU is forced to use the *odd* pipeline only and issue one instruction at a time.

Figure 25.3 shows the relation between the branch penalty and the number of cycles between the hint instruction and the branch instruction, when the branch

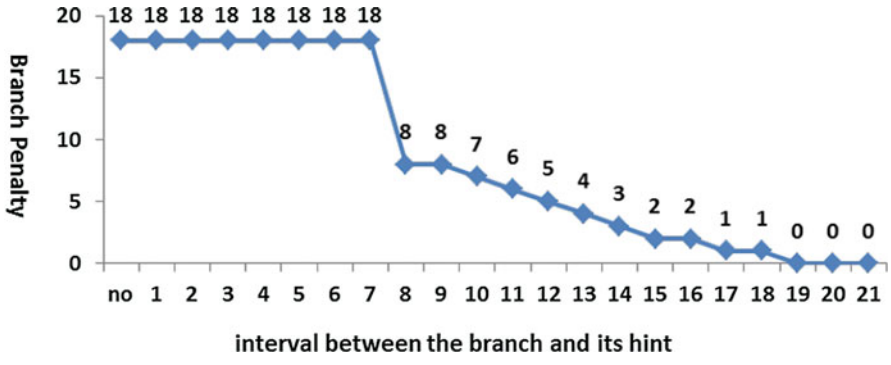


Fig. 25.3 The relation of the branch penalty, and the number of cycles between the hint instruction and the branch instruction, when the branch is predicted correctly

is correctly predicted. In other words, the branch is actually taken (since a hint instruction always loads the instruction at the target address of the taken branch). When the hint instruction is scheduled less than 8 cycles before the branch, the branch penalty is always 18 cycles. It implies the hint needs so much time to be properly set up and recognized when the branch starts to execute. **By default, the SPU always does not take branch prediction**. Therefore, without the hint, the SPU will keep predicting the incorrect direction for the branch and force the execution to pay the full branch penalty, i.e., resolving the target address and loading the instruction. The full branch penalty is measured as 18 cycles. As the interval between the hint and the branch is increased to be equal or greater than eight cycles (by inserting *lnop* instructions), the branch instruction is now aware of the existence of the hint. It still takes 18 cycles from the beginning of the hint instruction to the end of the branch instruction, since it also needs to resolve the branch target address and load the instruction, just like a branch instruction. However, by starting the entire process earlier, the hint instruction can hide some of the penalty, thanks to the instructions between the hint and the branch (the *lnop* instructions inserted), when the branch starts to execute. Notice the *lnop* instructions inserted are just placeholders for investigating the effect of the interval (between the hint and the branch) on the branch penalty. In the real execution, these will be replaced by the meaningful instructions. When the interval becomes equal or greater than 19 cycles, the branch penalty is completely eliminated. The branch penalty model in the SPU can be therefore built from the observation from the above experiment as follows:

$$Penalty_{correct}(l) \approx \begin{cases} 18, & if\ l < 8 \\ 18 - l, & if\ 8 \leq l < 19 \\ 0, & if\ l \geq 19 \end{cases} \tag{25.2}$$

where l denotes the number of cycles between the hint and the branch to be hinted.

Figure 25.4 shows the relation between the branch penalty and the number of cycles between the hint instruction and the branch instruction, when the branch is

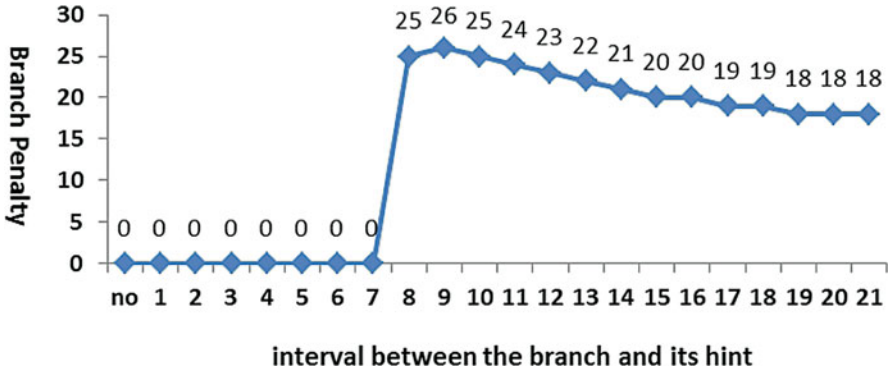


Fig. 25.4 The relation of the branch penalty, and the number of cycles between the hint instruction and the branch instruction, when the branch is mispredicted

mispredicted. In other words, the branch is not taken. When the interval is less than 8 cycles, the hint is not recognized at the branch, and it pays 18 cycles branch penalty just like before. However, when the interval is increased to be equal or greater than 8 cycles, the number of cycles spent from the beginning of the hint until the finish of the branch becomes greater than 18 cycles. This is because the branch instruction will start to wait for the incorrect target instruction to be loaded once it perceives the hint instruction. After the loading of the (incorrect) target instruction is completed, the branch instruction will start over, which will spend another 18 cycles. If the interval is further increased to be equal to or more than 19 cycles, the penalty becomes 18 cycles again, since the effect of misprediction will have been completely hidden by the time spent on executing the instructions between the hint and the branch, so that the branch will be executed as if the hint never happens and pays the 18 cycle penalty as if there is not any hint. The penalty of a branch misprediction thus can be modeled as follows:

$$Penalty_{incorrect}(l) \approx \begin{cases} 0, & if\ l < 8 \\ (18 - l) + 18 = 36 - l, & if\ 8 \leq l < 19 \\ 18, & if\ l \geq 19 \end{cases} \tag{25.3}$$

where l denotes the number of cycles between the hint and the branch to be hinted in the number of cycles.

25.1.1.4 Branch Hinting-Based Compilation

No Padding. When the number of cycles between the hint and the branch it will hint is smaller than a threshold value (eight in the SPU), the branch has to pay for the full branch penalty. In this case, we can insert NOP instructions (both *nop and lnop* instructions) to create a sufficiently large interval. Take Fig. 25.5 as an example.

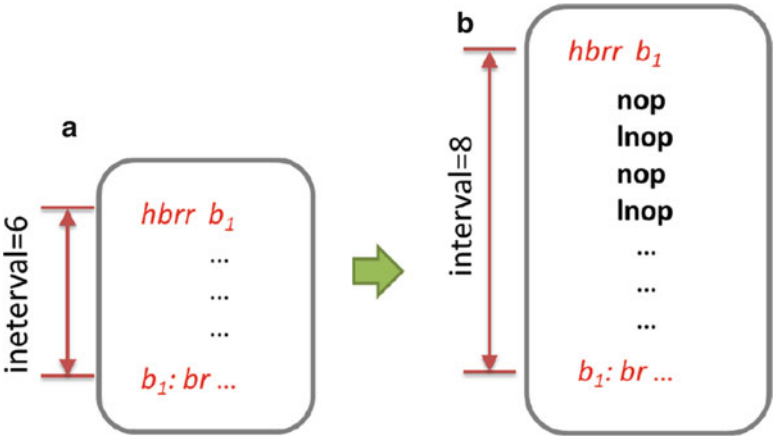


Fig. 25.5 NOP padding increases the interval between the hint instruction and the branch from 6 instructions/cycles in (**a**) to 8 instructions/cycles in (**b**) so that $b1$ can be hinted

Assume the branch *br* is taken and the hint instruction *hbrr* was originally executed six cycles earlier than *br*. According to the branch penalty model, the branch penalty is 18 cycles. After inserting two pairs of *nop/lnop* instructions, the interval becomes eight cycles, since each pair of *nop/lnop* can be executed in the *even/odd* pipeline at the same cycle, respectively, in the SPU. The branch penalty is therefore reduced to 10 cycles. The overall improvement is hence $18 - 10 - 2$ cycles, i.e., 6 cycles.

The SPU GCC compiler also provides a scheme for inserting *nop* (inserts both *nop* and *lnop* instruction) when a user-specified flag is enabled [7]. The SPU GCC inserts whenever the interval between a hint and the branch is not long enough. Carrying out NOP padding without deliberation may hurt performance sometimes.

To find out whether NOP padding is necessary under certain circumstances, we need a way to estimate the effect to the performance of applications if we insert the NOP instructions. Let l, n, p, and n_{NOP}, respectively, denote the interval between a hint and the branch to hint before NOP padding, the number of times the branch is executed, the taken probability of the branch, and the number of NOP instructions inserted. The branch penalties before the NOP padding can be calculated as follows:

$$Penalty_{no\_pad} = Penalty(l, n, p) \tag{25.4}$$

The branch penalties after the NOP padding can be calculated as follows,

$$Penalty_{pad} = Penalty(l + n_{NOP}, n, p) \tag{25.5}$$

For example, before the NOP padding, if the interval is smaller than eight cycles so that the hint is not recognized, then by applying the model of branch penalty introduced, we can get the branch penalty as $18 \times np \ + 18 \times n(1 - p) = 18n$.

When the branch instruction is executed in a loop, the corresponding hint instruction and the inserted NOP instructions must also be executed within the loop so to take effect at each iteration. Therefore, the number of times the inserted NOP instructions executed will be the same as the number of times the branch instruction is executed. Moreover, each pair of *nop* and *lnop* instructions can be executed in one cycle, thanks to the dual-issue pipeline in the SPU. Therefore, the overhead for executing the inserted NOP instructions can be modeled as the follows:

$$Overhead_{pad} = n(n_{NOP} + 1)/2 \tag{25.6}$$

So far, we have discussed the branch penalties before and after NOP padding and the extra overhead for the execution of the inserted NOP instructions. The impact of NOP padding on performance can be now modeled as follows:

$$Profit_{pad} = Penalty_{no\_pad} - Penalty_{pad} - Overhead_{pad} \tag{25.7}$$

Clearly, the NOP padding should be carried out only when the calculated number is positive.

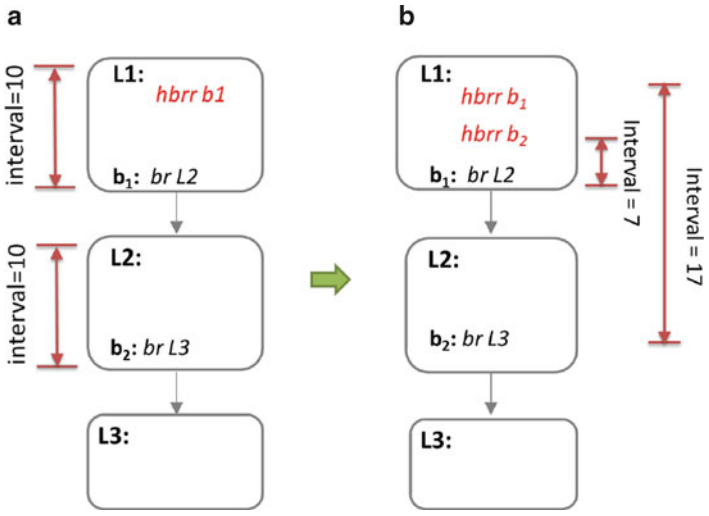


Fig. 25.6 Hint pipelining makes use of the minimum interval required to activate a hint instruction to insert extra hint instructions

Hint Pipelining. When two branches are close to each other, the hint instructions for the branches may interfere with each other. Figure 25.6a shows an example on how the SPU GCC compiler deals with such case. The SPU GCC compiler first tries to insert hints for both branches b_1 and b_2. However, when the hint instruction for b_2 has to be placed before b_1 to create a sufficiently large interval, it may overwrite the hint instruction for b_1. In this case, the GCC compiler estimates the priorities of the two branches, decides b_2 should be prioritized, and thus discards the hint for b_1. This problem is nevertheless not unsolvable, by prudently choosing the locations for both hints.

From the previous discussion, we have learned that a hint instruction will not be recognized by a branch instruction if it is executed within eight cycles before the branch. This gives us an opportunity to hint both branches. Figure 25.6b shows the method to hint two branches that are very close to each other. Although the hint instruction for b_2 is inserted before b_1, the interval between them is less than eight cycles. When b_1 is executed, the second hint is not recognized, so b_1 can be hinted correctly. However, when b_2 starts to execute later, the second hint will have been set up properly and take effect. With this approach, both b_1 and b_2 can be hinted. This method is called hint pipelining, as it "pipelines" hint instructions in the sense that the execution of the hint instructions are overlapped.

Again, to make sure this method is profitable, we need a cost model to find out the cost before applying this method and after applying this method. Assume in the given example in Fig. 25.6b that l_x denotes the number of instructions in the basic block L_x, p_x denotes the taken probability of the branch b_x, and n_x denotes the number of times b_x is executed. Keep in mind that we assume each instruction takes

one cycle, so l_x can be equally understood as the number of cycles that L_x takes to execute. The branch penalties before applying the hint pipelining method can be then calculated as follows:

$$\begin{aligned} Penalty_{no\_pipeline} = & \ Penalty(0, n_1, p_1) \\ & + (1 - p_1) \cdot Penalty(l_1 + l_2, n_2, p_2) \\ & + p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

The first term on the right-hand side is the branch penalty for b_1. Originally b_1 is not hinted, which can be viewed as if the interval is 0 cycle. The second and third term on the right-hand side are the branch penalties for b_2 when b_1 is not taken and when it is taken, respectively. When b_1 is not taken, b_2 will be hinted, and the interval between it and its hint is the sum of the number of instructions in both basic blocks L_1 and L_2; on the other hand, when b_1 is taken, b_2 will not be hinted, since the control flow will be diverted to a different basic block.

After hint pipelining is applied, both branches are hinted, although the interval between branch b_2 and its hint is decreased from $l_1 + l_2$ to $7 + l_2$ in the example. The branch penalties are changed as follows:

$$\begin{aligned} Penalty_{pipeline} = & \ Penalty(l_1, n_1, p_1) \\ & + (1 - p_1) \cdot Penalty(7 + l_2, n_2, p_2) \\ & + p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

Notice l_1 should be at least eight for this method to pan out, since otherwise the hint for b_1 will still not be recognizable.

The overhead of hint pipelining is the number of times the newly introduced hint instruction for b_1 is executed. Since the hint is inserted in basic block L_1, the number of times it is executed will be the same as b_1. The overhead is therefore as follows:

$$Overhead_{pipeline} = n_1 \tag{25.8}$$

The impact of hint pipelining on performance can be modeled as follows:

$$\begin{aligned} Profit_{pipeline} = & \ Penalty_{no\_pipeline} \\ & - Penalty_{pipeline} \\ & - Overhead_{pipeline} \end{aligned} \tag{25.9}$$

We should apply hint pipelining method only if the calculated number is positive.

Loop Restructuring. The NOP padding and hint pipelining methods are applicable when there is no loop or in the innermost loop. The loop restructuring

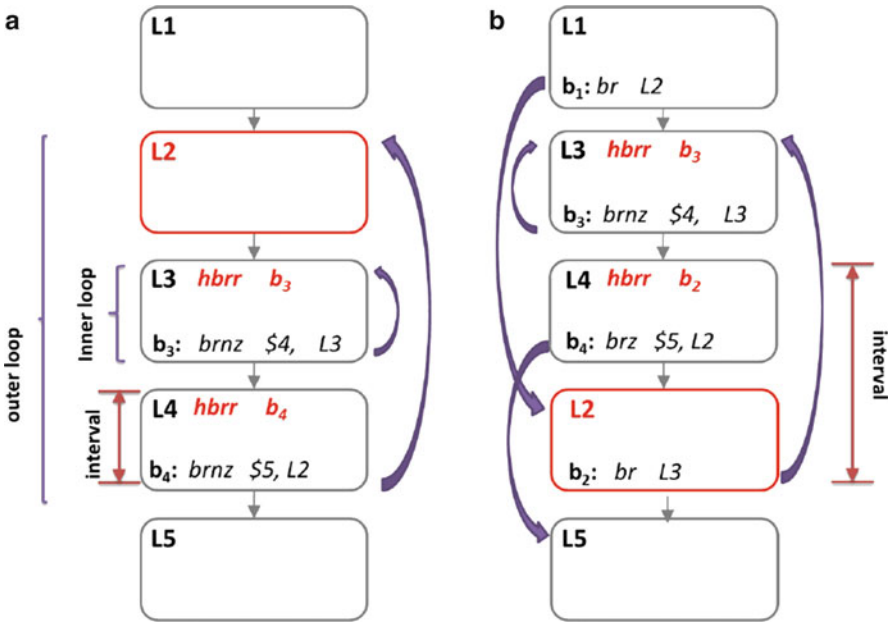


Fig. 25.7 Loop restructuring increases the leeway of hinting b_4 from l_4 in (**a**) to $l_2 + l_4$ in (**b**). Notice b_4 in (**a**) becomes b_2 in (**b**)

method, on the other hand, can be applied to outer loops in nested loops. This is done by altering the order of the basic blocks of the loops while keeping the semantic unchanged.

Figure 25.7 shows an example of nested loop restructuring method to reduce branch penalties in nested loops. Originally in Fig. 25.7a, b_4 is the condition of the outer loop, and the hint instruction for the branch b_4 is limited within the basic block L_4. This is because L_4 is preceded by a loop that consists of one basic block L_3. If the hint instruction for b_4 is inserted in any other basic block (earlier than L_4), it needs to either wait for the execution of all the iterations of L_3 (if the hint is inserted earlier than L_3), or it will be executed in every iteration of L_3 (if the hint is inserted in L_3). Neither case will lead to satisfactory performance. After restructuring as in Fig. 25.7b, L_2 is moved after L_4. To maintain the semantic, two unconditional branches from L_1 to L_2 and from L_2 to L_4 are introduced, respectively, and the condition of L_4 is modified accordingly. Such restructuring essentially turns b_2 into the condition of the outer loop. As a result, the possible location to insert a hint for b_2 is now increased to cover both L_4 and L_2.

Again, let l_x, p_x, and n_x, respectively, denote the number of instructions in the basic block L_x, the taken probability of the branch b_x, and the number of times b_x is executed. The branch penalties before restructuring of loops are as follows:

$$Penalty_{no\_reorder} = Penalty(l_3, n_3, p_3) + Penalty(l_4, n_4, p_4) \tag{25.10}$$

The branch penalties after restructuring the loops are as follows:

$$Penalty_{reorder} = 18 + Penalty(l_2 + l_4, n_2, p_2) + Penalty(l_3, n_3, p_3) + 18$$

Here the two 18s are the penalties for b_1 when entering the outer loop for the first time and for b_4 when exiting the outer loop when it is done, respectively.

The overhead of loop restructuring depends on the original size of basic block L_4. If it has less than eight instructions, then originally no hint can be made for b_4. After the restructuring, we will introduce a new hint in basic block L_4, which will be executed n_4 times. Otherwise, if originally there are more than eight instructions in L_4, no extra hint is introduced, and the overhead is zero. Therefore, the overhead for this method is as follows:

$$Overhead_{reorder} = \begin{cases} n_4, & if\ l_4 < 8 \\ 0, & otherwise \end{cases} \tag{25.11}$$

As a further optimization, some of the hint instructions can be promoted to be outside of a loop to avoid repeated computations. For example, Fig. 25.8a shows the code after promoting the hint for b_3 in the code given in Fig. 25.7a. After the promotion, the hint instruction only needs to be executed once while still

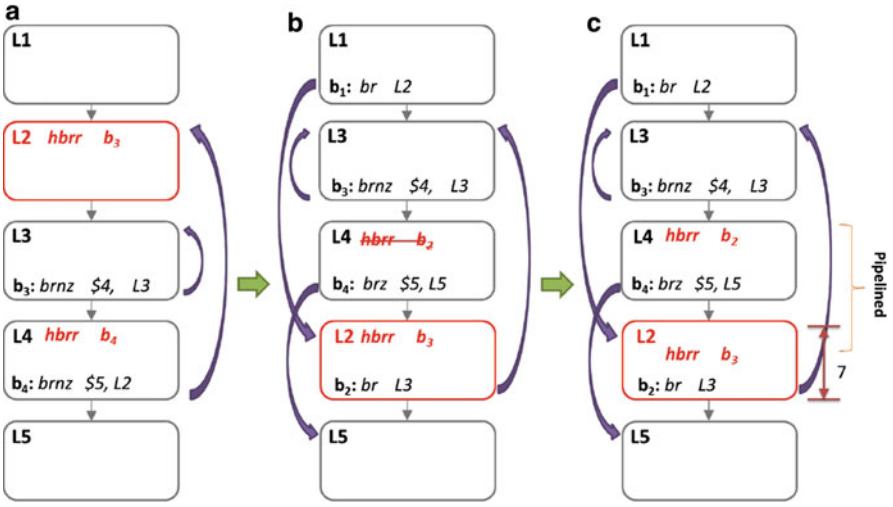


Fig. 25.8 The hint for b_3 is promoted to $L2$ from $L3$ in (**a**). This however may cause problems after loop restructuring as in (**b**), when the hint for b_3 overwrites the hint for b_2. Hint pipelining can be applied to enable both branches being hinted as in (**c**)

maintaining the semantics of the code. However, after the restructuring of nested loops as in Fig. 25.8b shows, the hint for b_3 overwrites the hint for b_2. This problem can be solved by applying the hint pipelining technique, as shown in Fig. 25.8c. Notice the promotion of a hint instruction should be applied only if the basic block the hint is promoted to itself does not have any taken branches, e.g., conditional or unconditional branches; otherwise, the promoted hint may interfere with the hints for the taken branches.

The three methods of reducing branch penalties – NOP padding, hint pipelining, and nested loop restructuring – can be combined and integrated into the compiler, as an optimization pass. The pass first restructures nested loops. It then traverses the Control-Flow Graph (CFG) of each function, in a bottom-up manner. That is to say, the pass first visits the bottom node (last basic block), and then recursively goes up along its predecessors. Once a branch is identified, the pass tries to promote its hint to its basic block whenever possible: if there is a branch that is likely taken in the predecessor, the traversal stops and the hint is inserted in the basic block the stop happens; otherwise, the pass keeps going up until it meets a basic block with any likely taken branch, or the basic block is the root of the CFG. Notice that a compiler with this pass enabled needs three extra parameters other than the input program, i.e., d, f, and s.

A microarchitecture-aware compiler is extremely important to improve the power saving and performance of execution in a software branch hinted processors. Figure 25.9 shows the performance improvement of the presented heuristic when being compared to the software branch hinting scheme provided by the SPU GCC compiler. The benchmarks that spend less than or equal to 20% of overall execution time for branches are considered with *low* branch penalty, while the others are considered as with *high* branch penalty. The heuristic outperforms the GCC scheme in every benchmark, and in general, the higher the ratio of branch penalty, the more the performance gain from applying the presented heuristic.

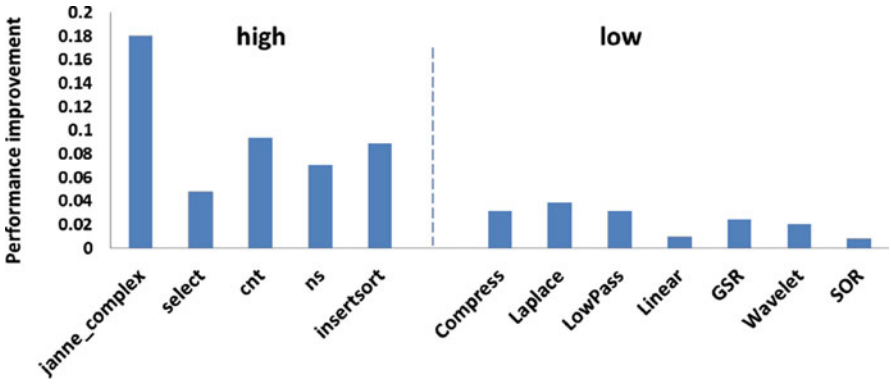


Fig. 25.9 Performance improvement of the presented heuristic is as much as 18% compared to the SPU GCC

25.1.2 Hardware-Aware Compilers for Design Space Exploration

Hardware-aware compilers are especially important for the design of embedded processors. At high level, the embedded processor design comprises of figuring out the microarchitectural configuration of the processor that will result in the best power and performance characteristics. Traditional DSE relies solely on simulation, as shown in Fig. 25.10. The same (compiled) code is measured on architectural models with different design parameters. The design parameters that yield the most desirable outcome are chosen. However, using the same code for different architectural variations may not guarantee fairness of the comparison, since the optimal code generated may vary as the design parameters changes. For example, loop tiling divides the iteration space into tiles or blocks to better fit the data cache. If we change the cache parameters, such as cache-line size or cache associativity, then we may need to change the size of each tile. Therefore, to be able to accurately explore the design space, a hardware-aware compiler should be included in the loop of DSE, so that every time the architectural design parameters are changed, the code generation should be adjusted accordingly, by compiling with the changed design parameters. Figure 25.11 shows the example of a framework of CIL DSE. CIL DSE is especially important in embedded systems, where the hardware-aware compiler can have a very notable impact on the power and performance characteristics of the processor.

In the rest of this subsection, we will study PBExplore – a framework for CIL DSE of partial bypassing in embedded processors. At the heart of the

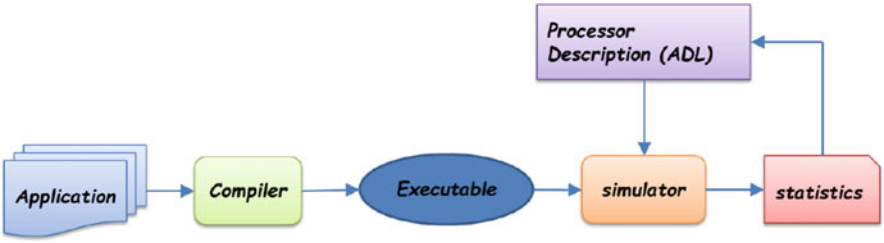


Fig. 25.10 Traditional DSE relies solely on simulation

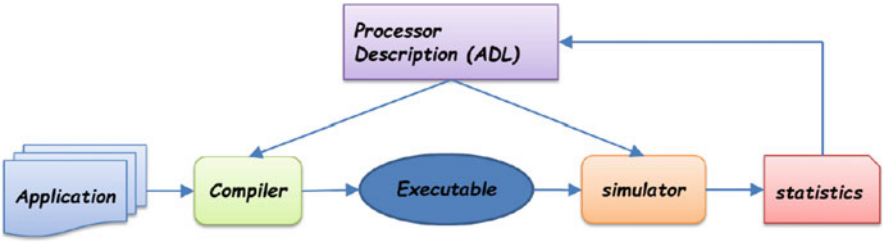


Fig. 25.11 CIL DSE includes the compiler in the loop of exploring best design parameters

PBExplore is a compiler that can generate high-quality code for a given partial bypassing configuration. Such a compiler can be used to explore different bypass configurations and discover the one that offers the best power, performance, cost, and complexity trade-offs.

25.1.2.1 The Case for Partial Bypassing

Pipelining is a widely used technique in modern processors to explore instruction-level parallelism and allow processors to achieve much higher throughput. However, the presence of hazards in the pipeline greatly impairs its value as they stall the pipeline and cause significant performance loss. Consequently, techniques are proposed to resolve the problems [10, 23]. Bypassing, also known as operand forwarding, is a popular solution to reduce data hazards. Bypassing adds additional datapaths and control logic to the processor so that the result of an operation can be forwarded to subsequent dependent operations even before it is written back to the register file.

While bringing in the great benefit, bypasses increase design complexity and may introduce significant overhead. Bypasses are often included in time-critical datapaths and therefore cause pressure on cycle time, especially the single cycle paths. This is particularly important in wide issue machines, where the delay may become more significant – due to extensive bypassing very wide multiplexors or buses with several drivers may be needed. Partial bypassing presents a trade-off between the performance, power, and cost of a processor and is therefore an especially valuable technique for application-specific embedded processors. Also, note that adding or removing bypasses or the bypass configuration of the processor does not affect its ISA.

25.1.2.2 Operation Latency-Based Schedulers Cannot Accurately Model Partial Bypassing

Traditionally, the retargetable compiler uses constant operation latency of each operation to detect and avoid data hazards [23]. The *operation latency* of an operation o is defined as a positive integer $ol \in I+$, such that if any data-dependent operation is issued more than ol cycles after issuing o, there will be no data hazards. When no bypassing or complete bypassing (the result of an operation can be forwarded at every stage of a pipeline once it is calculated and before it is committed to the memory system) is implemented in a pipeline, the operation latency is constant, and therefore the retargetable compiler can work perfectly. However, the presence of partial bypassing (the result of an operation can be forwarded only at some stage(s) but not all the stages of a pipeline) introduces variable operation latency and poses challenges for such a compiler.

To better understand the challenge of designing retargetable compilers in the presence of partial bypassing, let us first consider the differences of pipelines without bypassing, with complete bypassing, and with partial bypassing. Figure 25.12 illustrates the execution of an ADD operation in a simple five-stage pipeline without any bypassing. In the absence of any hazards, if the ADD operation is in F pipeline stage in cycle i, then it will be in OR pipeline stage in cycle i + 2. At this time, it

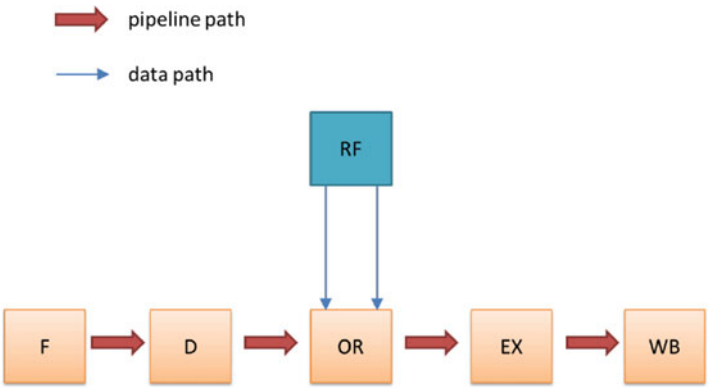


Fig. 25.12 A 5-stage processor pipeline with no bypassing

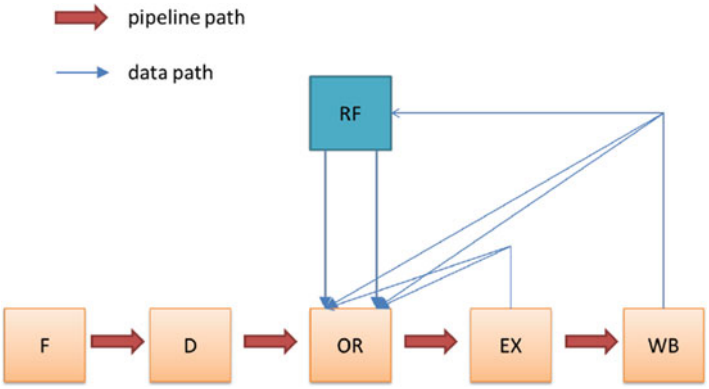


Fig. 25.13 A 5-stage processor pipeline with complete bypassing

reads the two source registers. The ADD operation then writes back the destination register in cycle i + 4, when it reaches the WB pipeline stage. The result of the ADD operation can be read from the register file in and after cycle i + 5. The operation latency of the ADD operation is three cycles $((i+5) - (i+2))$, so any instructions that are dependent on the result of the current instruction have to be scheduled at least three cycles later to avoid the data hazards. Figure 25.13 shows the pipeline with complete bypassing. The pipeline now includes forwarding paths from both execution (EX) and write back (WB) stages to both the operands of operand reading (OR) stage. The operation latency now becomes one cycle, since any dependent instructions scheduled one or two cycles after the ADD instruction can read its result from the bypasses, while those scheduled three or more cycles later can read the result from RF. Finally, we show an example of partial bypassing in Fig. 25.14. The pipeline only contains bypasses from EX (but not WB) stage to both the operands of OR pipeline stage. In this circumstance, scheduling a data-dependent operation one

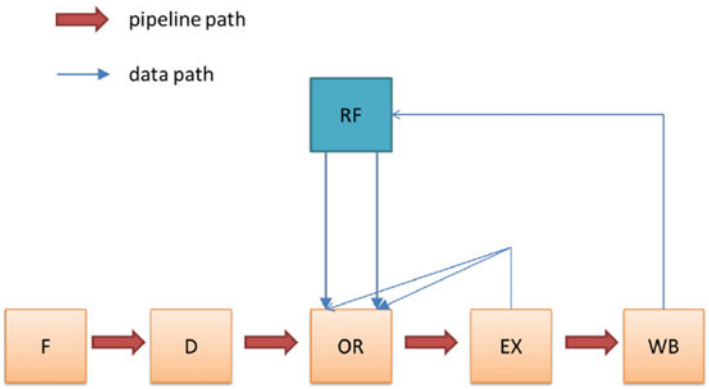


Fig. 25.14 A 5-stage processor pipeline with partial bypassing

or three cycles after the ADD will not result in a data hazard, since its result can be read either from EX pipeline stage via the bypasses or from the register file after the ADD operation writes back its result. However, if the data-dependent operation is scheduled two cycles after the scheduling ADD operation which is currently in WB stage, then it cannot do anything but wait, since no bypasses from WB are present. A data hazard happens. The operation latency of ADD in the partial bypassed pipeline in Fig. 25.14 is denoted by one, three, which means that scheduling a data-dependent operation one or three or more cycles after the schedule cycle of ADD will not cause any data hazard, but scheduling the data-dependent operation two cycles after the schedule cycle of ADD will cause a data hazard. The operation latency becomes nonconstant under partial bypassing. As a result, partial bypassing paralyzes the traditional retargetable compilers, which assumes a constant operation latency to detect pipeline hazards.

Without the accurate pipeline hazard detection technique, the retargetable compiler has to either conservatively assume no bypassing is present or aggressively assume that the pipeline is completely bypassed. However, both approaches will result in suboptimal code generated. To solve this problem, Operation Table (OT) [28] can be employed. An OT maintains the snapshot of the processor resources an operation uses in each cycle of its execution. It takes into consideration the (partial) bypassing in the pipeline and can therefore detect data hazards in advance even for partially bypassed processors. Besides, as the OT records at each cycle which processor resources are used, it is able to detect the structural hazards as well. As a result, an OT-based scheduler can accurately detect and avoid pipeline hazards and improve processor performance.

25.1.2.3 OT to Accurately Model the Execution of Operations in a Pipeline

In this subsection, we present the concept of OT that can accurately model the execution of operations in a processor pipeline, and later we will use them to

Table 25.2 Definition of the OT

OperationTable	:=	{ otCycle }
otCycle	:=	unit ros wos bos dos
ros	:=	ReadOperands { operand }
wos	:=	WriteOperands { operand }
bos	:=	BypassOperands { operand }
dos	:=	DestOperands { regNo }
operand	:=	regNo { path }
path	:=	port regConn port regFile

Table 25.3 The OT of ADD R1 R2 R3

1	F	
2	D	
3	OR	
	ReadOperands	
	R2	
		p1, C1, p6, RF
	R3	
		p2, C2, p7, RF
		p2, C5, p3, EX
	DestOperands	
	R1, RF	
4	EX	
	BypassOperands	
	R1	
		p3, C5, p2, OR
5	WB	
	WriteOperands	
	R1	
		p4, C3, p8, RF

develop a bypass-aware instruction scheduler. An OT describes the execution of one operation in the processor. Table 25.2 shows the grammar of an OT. Each entry, *otCycle*, in an OT describes the state of the operation in that execution cycle in the pipeline. *otCycles* are sorted in temporal order. Each *otCycle* records in that cycle the pipeline unit the operation is in (*unit*), the operands it needs to read (*ros*), write (*wos*), or bypass (*bos*), and the destination registers (*dos*) the operation may write to. Each operand (*operand*) is defined by the register number (*regNo*) and all the possible paths it may be transferred. Each possible path (*path*) consists of the ports (*port*), register connections (*regConn*), and the register file (*regFile*).

Table 25.3 shows the OT of executing an add operation, *ADD R1 R2 R3*, on the partially bypassed pipeline shown in Fig. 25.15. Without loss of the generality, assume by the time the add operation starts to execute, no hazards are present, so the add operation can be executed in five cycles, and consequently the OT of this

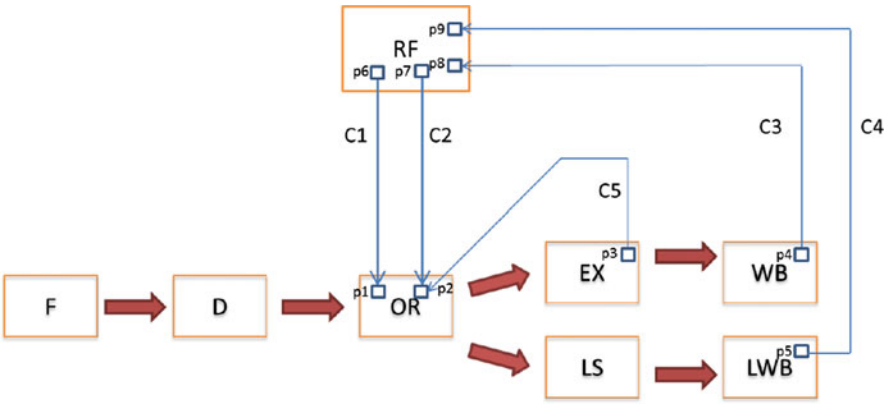


Fig. 25.15 An example partially bypassed pipeline

add operation contains five *otCycles*. The add operation is fetched to the *F* pipeline unit and decoded at *D* pipeline unit, respectively, during the first two cycles. In the third cycle, the add operation proceeds to *OR* pipeline unit and reads its source operands *R2* and *R3*. All the possible paths to read the each operand are included in the table. The first operand *R2* can be read only from the register file *RF* via connection *C1*, while the second operand *R3* can be read either from the register file *RF* via the connection *C2* from port *p7* to *p2* or from the pipeline unit *EX* via connection *C5* from port *p3* to port *p2*. In addition to the source operands, the destination operands *R1* are listed as well, and the dependent operations should be scheduled after accordingly. In the fourth cycle, the add operation is sent to the pipeline unit *EX* for execution. At the end of this cycle, the result of the operation is calculated and available for bypassing. The operation at the *OR* unit at that time can read the calculated result as its second operand via connection *C5*. In the last cycle, the operation is written back to *R1* from *WB* pipeline unit to the register file *RF* via connection *C3*.

There may be multiple paths to read each operand in the presence of bypasses. As an example, the Intel XScale processor provides seven possible bypasses for each operand, in addition to the register file. The OT of an operation lists all the possible paths to read each operand. As a result, the OT may potentially have to store eight paths (seven from bypasses plus one from the register file) for each an operand to read. To prevent such superfluity to consume too much space in the OT, the concept of Bypass Register File (BRF) is introduced. A BRF is essentially a virtual register file that serves as a temporary storage for each operand having bypasses. All the values bypassed to the operand are first written to the BRF and then read by the operation. A value from the bypass must be read in the same cycle once the value is calculated; therefore, each value can exist only for one cycle in the BRF. Each operand needs only one BRF to accept values from all the bypasses that attach to it. As a result, the space consumed by an OT can be greatly reduced.

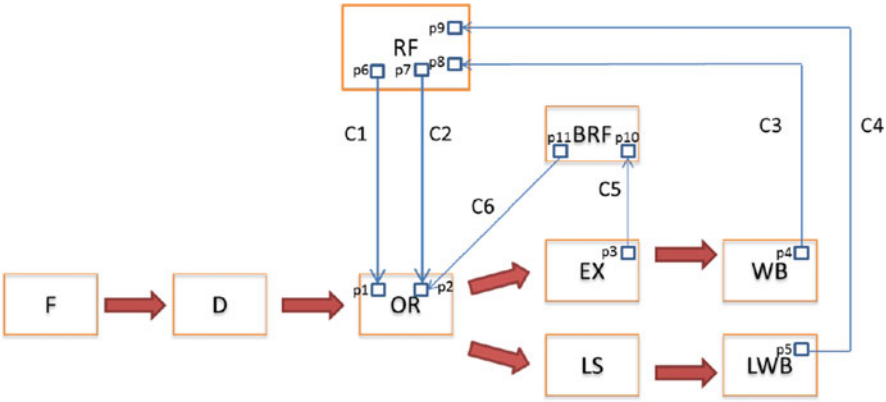


Fig. 25.16 An example partially bypassed pipeline with BRF

Table 25.4 The OT of ADD R1 R2 R3 with BRF

1	F	
2	D	
3	OR	
	ReadOperands	
	R2	
		p1, C1, p6, RF
	R3	
		p2, C2, p7, RF
		p2, C6, p11, BRF
	DestOperands	
	R1, RF	
4	EX	
	BypassOperands	
	R1	
		p3, C5, p10, BRF
5	WB	
	WriteOperands	
	R1	
		p4, C3, p8, RF

Figure 25.16 shows the processor pipeline with a BRF. The bypassed result from the *EX* unit to the second source operand of the *OR* unit is first written to the BRF via connection *C5*. The *OR* unit then reads the value either from the BRF via connection *C6* or from actual register file *RF* via connection *C2*. Table 25.4 shows the OT of the operation *ADD R1 R2 R3* in the pipeline with the BRF in Fig. 25.16. The only differences are that the second source operand of *OR* unit can be read from either the actual register file *RF* or the virtual register file *BRF*, and the result of the *EX* unit is now first bypassed to the *BRF* instead of directly to the second source operand of the *OR* unit.

Detecting Pipeline Hazards Using OT

To illustrate the power OT possesses in detecting pipelining hazards, let us consider an example of applying OT-based scheduling in the pipeline in Fig. 25.16 on the three operations as follows:

MUL R1 R2 R3 ($R1 \leftarrow R2 \times R3$)
ADD R4 R2 R3 ($R4 \leftarrow R2 + R3$)
SUB R5 R4 R2 ($R5 \leftarrow R4 - R2$)

Assume both SUB and ADD operations take one cycle in the *EX* stage and the MUL operation spends two cycles in the same stage. In addition, all the pipeline resources are available initially. Therefore, when the MUL operation is scheduled at the first cycle, there will be no hazards. Table 25.5 shows the state of the machine after MUL is scheduled.

We then try to schedule ADD in the next cycle. However, a resource hazard will happen, since the *EX* pipeline unit is still busy executing the second cycle of *MUL* operation. Table 25.6 shows the state of the processor pipeline after scheduling ADD in the second cycle. A resource hazard is detected when the fourth *otCycle* of ADD is tried in the fifth cycle, so the *otCycle* should be not be scheduled at this cycle.

At this point, if we keep scheduling the SUB operation in the third cycle, a data hazard will be detected. The SUB operation needs to read the value of the first operand *R4*, which is calculated by the previous ADD operation. However, the

Table 25.5 Pipeline states after scheduling MUL R1 R2 R3 in Cycle 1

Cycle	Busy resources	!RF	BRF
	Operation1		
1.	F	–	–
2.	D	–	–
3.	OR, p1, C1, p6, p2, C2, p7	–	–
4.	EX	R1	–
5.	EX, p3, C4, p10	R1	R1
6.	WB, p4, C3, p8	R1	–
7.		–	–

Table 25.6 Pipeline states after scheduling ADD R4 R2 R3 in Cycle 2

Cycle	Busy resources		!RF	BRF
	Operation1	Operation 2		
1.	F		–	–
2.	D	F	–	–
3.	OR, p1, C1, p6, p2, C2, p7	D	–	–
4.	EX	OR, p1, C1, p6, p2, C2, p7	R1	–
5.	EX, p3, C4, p10	Resource hazard	R1 R4	R1
6.	WB, p4, C3, p8	EX, p3, C4, p10	R1 R4	R4
7.		WB, p4, C3, p8	R4	–
8.			–	–

Table 25.7 Pipeline states after scheduling SUB R5 R4 R2 in cycle 3

Cycle	Busy resources			!RF	BRF
	Operation1	Operation 2	Operation 3		
1.	F			–	–
2.	D	F		–	–
3.	OR, p1, C1, p6, p2, C2, p7	D	F	–	–
4.	EX	OR, p1, C1, p6, p2, C2, p7	D	R1	–
5.	EX, p3, C4, p10	Resource hazard	Data hazard	R1 R4	R1
6.	WB, p4, C3, p8	EX, p3, C4, p10	Data hazard	R1 R4	R4
7.		WB, p4, C3, p8	Data hazard	R4	–
8.			OR, p1, C1, p6, p2, C2, p7	R5	–
9.			EX, p3, C4, p10	R5	R5
10.			WB, p4, C3, p8	R5	–
11.				–	–

bypass in Table 25.5 is from *EX* pipeline unit to the second operand in *OR* pipeline unit, so there will not be any available path for this operand to be transferred at the time the SUB operation enters to the *EX* pipeline unit. The data hazard is resolved in the eighth cycle. Table 25.7 shows that the state of the processor pipeline after SUB is scheduled in the third cycle. After the scheduling of the SUB operation, all the operations are successfully scheduled with both data and resource hazards detected, even in the presence of partial bypassing.

25.1.2.4 List Scheduling Algorithm Using OT

In the presence of partial bypassing, the operation latency of an operation is not sufficient to avoid all the data hazards – OT is needed. The traditional list scheduling algorithm can be very easily modified by using OT. Figure 25.17 shows the list scheduling algorithm that uses OT for pipeline hazard detection. The *DetectHazard* function (line 10) and the *AddOperation* function (line 13) are two functions that are based on OT. The *DetectHazard* function checks if scheduling all *otCycles* of the operation v starting from the machine cycle t will cause any hazards. Once the scheduler finds the earliest available machine cycle, it calls *AddOperation* function to schedule operation v in cycle t.

Experiments are performed in the Intel XScale microarchitecture to verify the capability of the OT-based scheme. Figure 25.18 shows the pipeline of XScale architecture. The experimental setup is shown in Fig. 25.19. Each benchmark application is first compiled with the GCC cross compiler for Intel XScale processor. The OT-based scheduling is then applied to each basic block of the original program to generate the executable again. The two versions of executable files are then run on the XScale cycle-accurate simulator, respectively. Performance is measured as the number of cycles spent on executing applications. The improvement of performance is measured as $(gccCycles - otCycles) * 100/gccCycles$, where $gccCycles$ is

Fig. 25.17 List scheduling algorithm using OT

ListScheduleUsingOTs(V)
01: $U = V - v_0; F = \varphi; S = v_0$

/* initialize */
02: **foreach** $(v \in V)$
03: $schedTime[v] = 0$
04: **endFor**

/* list schedule */
05: **while** $(U \neq \varphi)$
06: $F = \{v | v \in U, parents(v) \subset S\}$
07: $F.sort()$ /* some priority function */
08: $v = F.pop()$
09: $t = MAX(schedTime(p)),\ p \in parents(v)$
10: **while** $(DetectHazard(machineState, v.OT, t))$
11: $t++$
12: **endWhile**
13: $AddOperation(machineState, v.OT, t)$
14: $schedTime[v] = t$
15: **endWhile**

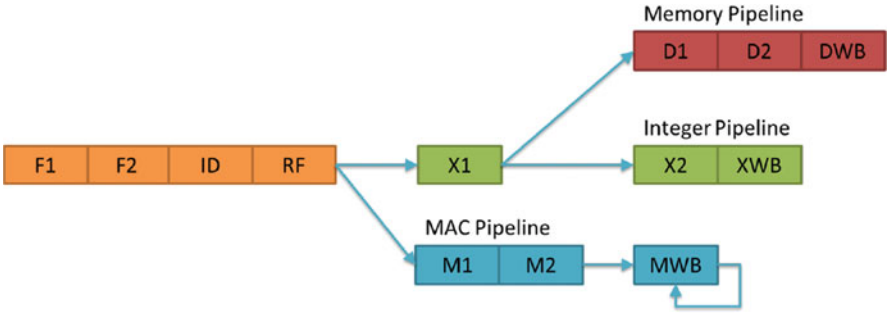
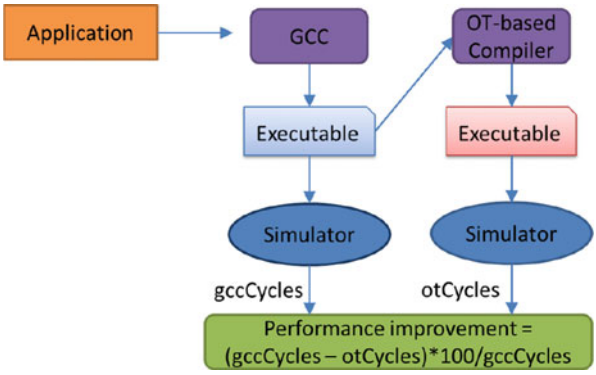


Fig. 25.18 The pipeline in XScale

Fig. 25.19 Experimental setup



the number of cycles spent on running an applications compiled with GCC compiler, and $otCycle$ is its counterpart with OT-based scheduling. Figure 25.20 shows the details. The OT-based scheme improves the performance over the GCC compiler by up to 20%.

25.1.2.5 CIL Partial Bypass Exploration

With effectiveness of OT-based scheme, we can further make use of it to explore the design space of partial bypassed processors. PBExplore, a CIL Framework for DSE of partial bypassing in processors is proposed to accommodate the need. The compiler in the PBExplore takes as input bypass configuration, as shown in Fig. 25.21. A bypass configuration describes the pipeline stage each individual bypass starts (source), and the operand that can consume the value the bypass transfers (destination). The binary generated by the bypass-aware compiler is

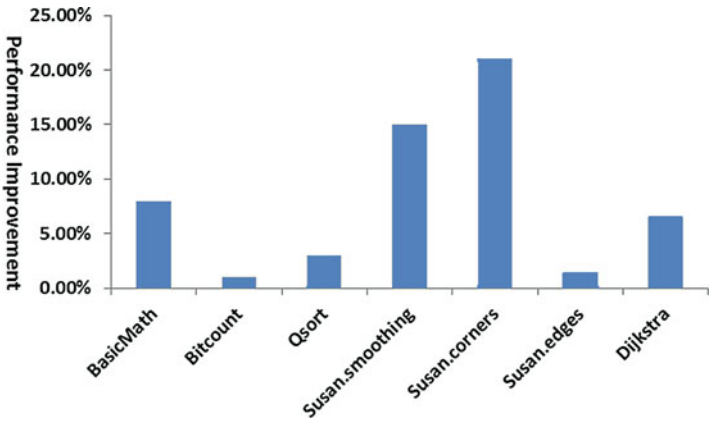


Fig. 25.20 Performance improvement of the compiler with OT-based scheduling over the GCC compiler

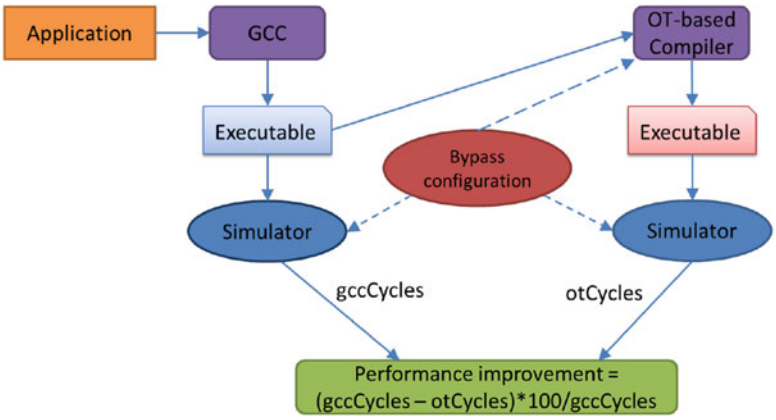


Fig. 25.21 PBExplore: A CIL framework for partial bypass exploration

then run on a cycle-accurate simulator that is parameterized on the same bypass configuration. The simulator then dumps the estimations of cycles of execution, area, and power consumption.

PBExplore can effectively guide designers to use the best design decisions and avoid suboptimal design decisions that may happen in simulation-only DSE. Figures 25.22, 25.23 and 25.24 show, respectively, the change of execution cycles when only the X-bypasses (enabling bypassing of the pipeline stages in the integer pipeline in Fig. 25.18), D-bypasses (enabling bypassing of the pipeline stages in the memory pipeline), and M-bypasses (enabling bypassing of the pipeline stages in the Multiply-Accumulator (MAC) pipeline) are varied, respectively, while the other two bypasses are fixed. Figure 25.22 shows that while the execution cycles for configurations < X2 X1 >, < XWB X1 > and < XWB X2 > are similar under

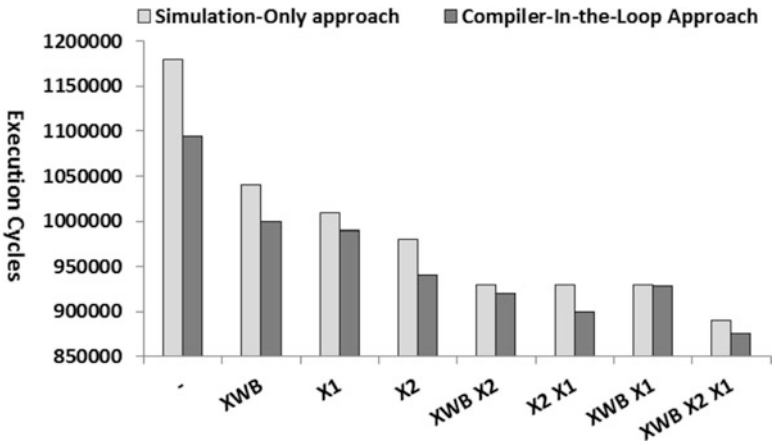


Fig. 25.22 X-bypass exploration for the bitcount benchmark

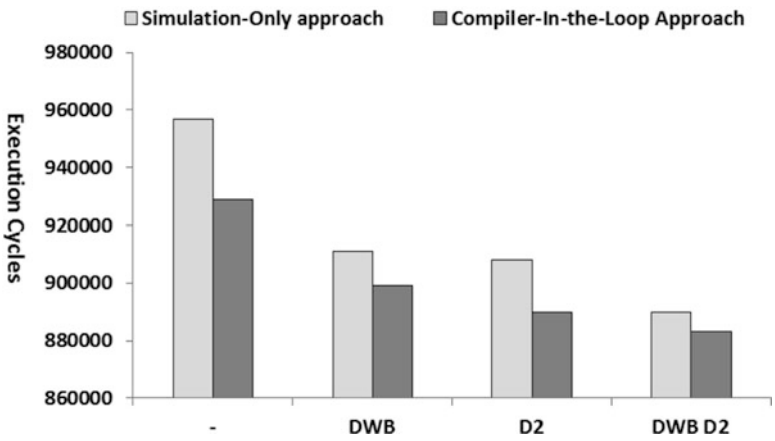


Fig. 25.23 D-bypass exploration for the bitcount benchmark

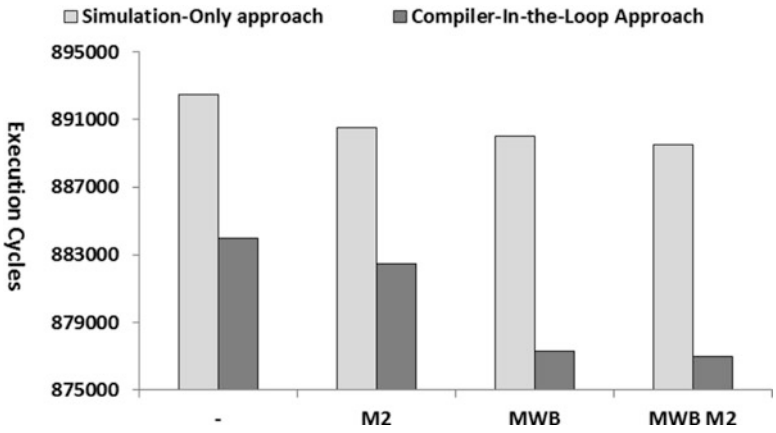


Fig. 25.24 M-bypass exploration for the bitcount benchmark

simulation-only approach, the bypass-sensitive compiler is able to identify < X2 X1 > the best among the three choices. If designers choose to have only two bypasses in the processor, then they would have made the wrong choice based on simulation solely. Similarly, Fig. 25.23 shows that when there is only one bypass, bypassing *D2* pipeline stage is a better choice than bypassing *DWB* according to PBExplore, while the simulation-only approach may mislead designers to bypass either *D2* or *DWB*. Similar observations can be found for the M-bypass exploration in Fig. 25.24 and the D-bypass exploration in Fig. 25.23.

25.1.3 Conclusions

Embedded systems are ubiquitous in our daily life, ranging from portable music players to real-time control systems in space shuttles. The diversity of embedded applications eventually boils down to multidimensional design constraints on embedded systems. To meet these constraints, embedded processors often feature unique design parameters, several missing features, and often quite quirky designs. For these embedded processors, the compiler often has a very significant impact on the power and performance characteristics of the processor – and therefore hardware-aware compilers are most useful and effective for embedded processors. Hardware-aware compilers take the microarchitectural description of the processor into account in addition to the application code in order to compile. There are two main use-cases for hardware-aware compilers. The first one is the traditional use, i.e., as a production compiler for an embedded processor. In addition to this, a hardware-aware compiler can be used to design an efficient embedded processor. The hardware-aware compiler enables the CIL DSE of the microarchitectural space of the processor, which takes into consideration the effects compilers have on the power consumption and performance of the processor. We demonstrate these two

uses by presenting a compiler technique to significantly alleviate branch penalties in processors without hardware branch prediction in Sect. 25.1.1, and a OT-based compiler technique that can be used to improve the performance and to help the design of processors with partial bypassing in Sect. 25.1.2. The experimental results corroborate the importance of hardware-aware compilation.

References

1. Bala V, Rubin N (1995) Efficient instruction scheduling using finite state automata. In: Proceedings of the 28th annual international symposium on microarchitecture, pp 46–56. doi:10.1109/MICRO.1995.476812
2. Ball T, Larus JR (1993) Branch prediction for free. In: Proceedings of PLDI. ACM, New York, pp 300–313. doi:10.1145/155090.155119
3. Chen T, Raghavan R, Dale JN, Iwata E (2007) Cell broadband engine architecture and its first implementation – a performance view. IBM J Res Dev 51(5):559–572. doi:10.1147/rd.515.0559
4. Dual-Core Intel Itanium Processor 9000 and 9100 Series (2007). http://download.intel.com/design/itanium/downloads/314054.pdf
5. Flachs et al B (2006) The microarchitecture of the synergistic processor for a cell processor. IEEE Solid-State Circuits 41(1):63–70
6. Fog A (2008) The microarchitecture of Intel and AMD CPUs
7. GNU Toolchain 4.1.1 and GDB for the Cell BE's PPU/SPU. http://www.bsc.es/plantillaH.php?cat_id=304
8. Grun P, Dutt N, Nicolau A Memory aware compilation through accurate timing extraction. In: Proceedings of the 37th annual design automation conference, DAC'00. ACM, New York, pp 316–321 (2000). doi:10.1145/337292.337428
9. Grun P, Dutt N, Nicolau A (2000) MIST: an algorithm for memory miss traffic management. In: IEEE/ACM international conference on computer aided design, ICCAD-2000, pp 431–437. doi:10.1109/ICCAD.2000.896510
10. Grun P, Halambi A, Dutt N, Nicolau A (2003) RTGEN-an algorithm for automatic generation of reservation tables from architectural descriptions. IEEE Trans Very Large Scale Integr (VLSI) Syst 11(4):731–737. doi:10.1109/TVLSI.2003.813011
11. Halambi A, Grun P, Ganesh V, Khare A, Dutt N, Nicolau A (1999) EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In: Design, automation and test in Europe conference and exhibition 1999. Proceedings, pp 485–490. doi:10.1109/DATE.1999.761170
12. Hoffmann A, Schliebusch O, Nohl A, Braun G, Wahlen O, Meyr H (2001) A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. In: Proceedings of the 2001 IEEE/ACM international conference on computer-aided design, ICCAD'01. IEEE Press, Piscataway, pp 625–630
13. https://gcc.gnu.org/ (2007)
14. IBM: Cell Broadband Engine Programming Handbook including PowerXCell 8i. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED
15. Intel Corporation. Intel XScale(R) Core: Developer's Manual. http://www.intel.com/design/iio/manuals/273411.htm
16. Keutzer K, Malik S, Newton A (2002) From ASIC to ASIP: the next design discontinuity. In: IEEE international conference on computer design: VLSI in computers and processors, 2002. Proceedings, pp 84–90. doi:10.1109/ICCD.2002.1106752
17. Kondo M, Kobyashi H, Sakamoto R, Wada M, Tsukamoto J, Namiki M, Wang W, Amano H, Matsunaga K, Kudo M, Usami K, Komoda T, Nakamura H (2014) Design and evaluation of

fine-grained power-gating for embedded microprocessors. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1–6. doi:10.7873/DATE.2014.158
18. Kongetira P, Aingaran K, Olukotun K (2005) Niagara: a 32-way multithreaded sparc processor. IEEE Micro 25(2):21–29. doi:10.1109/MM.2005.35
19. Lattner C (2002) LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana. See http://llvm.cs.uiuc.edu
20. Leupers R (2000) Code generation for embedded processors. In: The 13th international symposium on system synthesis, 2000. Proceedings, pp 173–178. doi:10.1109/ISSS.2000.874046
21. Lowney PG, Freudenberger SM, Karzes TJ, Lichtenstein WD, Nix RP, O'Donnell JS, Ruttenberg JC (1993) The multiflow trace scheduling compiler. J Supercomput 7:51–142
22. Lu J, Kim Y, Shrivastava A, Huang C (2011) Branch penalty reduction on IBM cell SPUs via software branch hinting. In: Proceedings of CODES+ISSS, pp 355–364
23. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco
24. Park D, Lee J, Kim NS, Kim T (2010) Optimal algorithm for profile-based power gating: a compiler technique for reducing leakage on execution units in microprocessors. In: 2010 IEEE/ACM international conference on computer-aided design (ICCAD), pp 361–364. doi:10.1109/ICCAD.2010.5653652
25. Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K (1997) A case for intelligent RAM. IEEE Micro 17(2):34–44. doi:10.1109/40.592312
26. Proebsting TA, Fraser CW (1994) Detecting pipeline structural hazards quickly. In: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'94. ACM, New York, pp 280–286. doi:10.1145/174675.177904
27. Roy S, Katkoori S, Ranganathan N (2007) A compiler based leakage reduction technique by power-gating functional units in embedded microprocessors. In: 20th international conference on VLSI Design, 2007. Held jointly with 6th international conference on embedded systems, pp 215–220. doi:10.1109/VLSID.2007.10
28. Shrivastava A (2006) Compiler-in-loop exploration of programmable embedded systems. Ph.D. thesis, Donald Bren School of Information and Computer Sciences
29. Shrivastava A, Issenin I, Dutt N (2005) Compilation techniques for energy reduction in horizontally partitioned cache architectures. In: Proceedings of the 2005 international conference on compilers, architectures and synthesis for embedded systems, CASES'05. ACM, New York, pp 90–96. doi:10.1145/1086297.1086310
30. Siska C (1998) A processor desription language supporting retargetable multi-pipeline DSP program development tools. In: Proceedings of the 11th international symposium on system synthesis, ISSS'98. IEEE Computer Society, Washington, DC, pp 31–36
31. Trimaran. http://www.trimaran.org/
32. Wagner TA, Maverick V, Graham SL, Harrison MA (1994) Accurate static estimators for program optimization. In: Proceedings of the ACM SIGPLAN 1994 conference on programming language design and implementation, PLDI'94. ACM, New York, pp 85–96. doi:10.1145/178243.178251
33. Wu Y, Larus JR (1994) Static branch frequency and program profile analysis. In: Proceedings of the 27th annual international symposium on Microarchitecture. ACM, New York, pp 1–11. doi:10.1145/192724.192725
34. Zivojnovic V, Pees S, Meyr H (1996) LISA-machine description language and generic machine model for HW/SW co-design. In: Workshop on VLSI signal processing, IX, pp 127–136. doi:10.1109/VLSISP.1996.558311