Parallelizing Compilation through Load-Time Scheduling for a Superscalar Processor Family

Michael Hußmann
International Graduate School
of Dynamic Intelligent Systems
University of Paderborn
Paderborn, Germany
michaelh@upb.de

Michael Thies
Faculty of Computer Science,
Electrical Engineering and
Mathematics
University of Paderborn
Paderborn, Germany
mthies@upb.de

Uwe Kastens
Faculty of Computer Science,
Electrical Engineering and
Mathematics
University of Paderborn
Paderborn, Germany
uwe@upb.de

ABSTRACT

Superscalar processors improve the execution time of sequential programs by exploiting instruction-level parallelism (ILP). The efficiency of parallelization at run-time can be increased through an additional scheduling phase for a concrete target machine in the compiler. But if the target machine is not known at compile-time, scheduling must be deferred to a later phase immediately before program execution. In this paper we present a novel technique, which prepares parallelization at compile-time and performs scheduling at load-time of a program. Our approach called *CALS* (Code Annotations for Load-time Scheduling) uses proof-carrying code techniques for scheduling in linear time by using a new algorithm. Additionally, the closely related task of register allocation is split between compile-time and load-time of a program. *CALS* achieves improvements of up to 23.8% over simple compilation without scheduling. It obtains results comparable to conventional list scheduling or even outperforms it by up to 12.4%.

1. INTRODUCTION

Superscalar processors improve the execution time of sequential programs by exploiting instruction-level parallelism (*ILP*). A *dispatcher* issues machine instructions to several functional units, where they are executed in parallel. But a *dispatcher* can re-arrange instructions only in a very constrained context of e.g. 4 consecutive operations in case of *PowerPC* hardware [29].

Stümpel et al. [28] demonstrated that an extra scheduling phase in the compiler can increase efficiency of the *dispatcher* in a *PowerPC 604* [20]. Scheduling techniques at compile-time consider a larger analysis context than *dispatcher* hardware at run-time, e.g. a basic block or a loop body. Additionally, more precise information about data dependencies between operations, especially memory accesses, are available due to prior program analysis.

Scheduling in a compiler also accounts for the types and numbers of functional units in the processor and hence optimizes a program for a concrete target machine. On the other hand, there exist so-called families of processors, which all use the same instruction set and encoding, but differ in types, numbers and effectiveness of their parallel execution units. In principle, all processors of a family can execute parallelized machine code, but maximum performance is only achieved for the processor model that was used as target machine during scheduling.

For mobile code in a heterogenous network with processors of one family, the precise machine model is not known at compile-time. Hence scheduling must be deferred to a later phase immediately before program execution.

In this paper we present a novel technique for increasing performance of applications on the superscalar *PowerPC* processors: Parallelization of machine code is prepared at compile-time, while scheduling for a concrete target machine is performed at load-time of a program. Our *CALS* approach uses *Proof-Carrying Code* (*PCC*) techniques [24]. The compiler produces code annotations, which enable very fast scheduling immediately before execution. In fact, our scheduling algorithm needs only linear time, while other scheduling techniques such as *list scheduling* [18] etc. have at least quadratic complexity. Additionally, the closely related task of register allocation is split between compile-time and load-time.

The rest of the paper is organized as follows: Section 2 discusses work related to our approach. Section 3 motivates the structure of our system as well as its compiler and scheduler. Section 4 outlines the major aspects towards preparation of scheduling at compile-time. Section 5 presents the structure of our *load-time scheduler* and the novel scheduling algorithm. Section 6 discusses the evaluation of our approach including key data such as speed-up and size of annotations. Finally, section 7 concludes the paper.

2. RELATED WORK

The related work is separated into two parts: First, we discuss five supposedly similar approaches that adapt program code to the target machine at run-time and emphasize the differences to our idea. Then we present several computer architectures that reconfigure the target machine for each program and compare them with our approach.

2.1 Selected Topics

Dynamo [2] optimizes the machine code of a program at run-time. It is an interpreter, which determines frequently executed code fragments and stores optimized versions in a fragment cache. If possible, the interpreter executes machine code from the fragment cache to improve execution time. Otherwise, the slow interpreting mechanism is used. Hence, performance is poor when starting a program, but improves during run-time. In contrast to our approach, the optimization at run-time is not prepared by a compiler.

`dcg` [10] is a retargetable system for dynamic code generation. The input is specified in the intermediate language *IR*, which is also used for the `lcc` compiler [13]. At run-time, the intermediate code is transformed into executable code for the concrete target machine by the code generator `burg` [14]. To simplify code specification *VCODE* [9] was developed, which accepts assembly language for an ideal model of a *RISC* processor. Like our approach, dynamic code generation offers high flexibility by creating executable machine code only at run-time. But it is not focused on scheduling and propagates a high computing effort from compile-time to run-time.

Our *CALS* is a concrete instance of a mobile code application. Another interesting example are *slim binaries* [12], which contain compact and architecture-neutral intermediate code. In contrast to a *fat binary*, that includes different versions of a program, the same intermediate code is transformed into specific native machine code on the target system. But again major parts of the compilation must be done immediately before or during program execution.

The motivation for *dynamic rescheduling* [6] is the lack of object code compatibility across generations of a *VLIW* architecture. Unlike superscalar processors, object code for *VLIW* machines consists of large instruction words, which specify multiple and independent scalar instructions for each functional unit. Hence, a transformation of *VLIW* code mainly has to conduct further scheduling. Dynamic rescheduling applies a scheduling during first-time page faults and thereby adapts object code to a concrete target machine. In contrast to our *load-time scheduler*, the rescheduling is not prepared at compile-time.

Binary translation [27] is a generalization of dynamic rescheduling and refers to object code transformation between different architectures. One example is *DAISY* [8], which emulates the *PowerPC* family on a *VLIW* machine. The famous *Crusoe* processors [22] execute *Intel x86* code and combine high performance with low power consumption. Such a processor consists of a *VLIW* machine and a software layer, which transforms *x86* code into *VLIW* code at run-time by using a *Code Morphing Software* [7]. Like dynamic rescheduling, all mentioned approaches do not prepare scheduling at an earlier time instant.

2.2 Reconfigurable Architectures

Reconfigurable hardware offers a promising extension of our approach: Currently, the *load-time scheduler* adapts the abstract machine code to a concrete and static target machine. In the future, we plan to extend our approach to reconfigurable architectures, where the target machine adapts itself to the code also. For example, a superscalar *RISC* processor could merge some of its functional units into *SIMD*-like structures on demand. By these means, *ILP* could be exploited more efficiently and hence execution time could be improved. In the following we discuss selected reconfigurable architectures and compare them with our approach.

Garp [17, 4] and *MorphoSys* [26] combine a *RISC* processor with a reconfigurable array to accelerate loops. Both approaches suffer from communication bottlenecks due to transfers between the main processor and the reconfigurable hardware at the entrance and exit of loops. Additionally, the programmer must manually partition source code (*MorphoSys*) or generate hardware configurations (*Garp*) to take advantage of those systems. Our system accepts *ANSI C* code and requires no further administration by the user.

Chimaera [16] overcomes the bottleneck of systems like *Garp* or *MorphoSys* by integrating a reconfigurable functional unit (*RFU*) into the host processor itself. The intention of the *RFU* is to customize the instruction set for a certain application. Hence, the compiler combines multiple adjacent operations into a single *RFU* instruction. The *RFU* is a cache for instructions, which have recently been executed or might be needed soon. To use instructions in the *RFU*, the application code includes calls to the *RFU* by means of special operations. In contrast to our approach, the *Chimaera* compiler does not utilize *ILP*, but optimizes the instruction set of the target machine.

DISC [25] was designed for high performance real-time systems, which are characterized by stochastically occuring interrupts as well as high throughput requirements. It uses dynamic instruction stream interleaving, i.e. the next instruction to be executed is dynamically selected from several simultaneously active streams. In contrast to our *load-time scheduler*, which is presented in 3.2, *DISC* operates completely at run-time.

OneChip [30] tightly couples a *RISC* processor with reconfigurable logic by integrating an *RFU* into the processor's pipeline. It was designed to speed up computing-intensive application in the multimedia or *DSP* domain on the loop-level. But like *Garp* and *MorphoSys*, *OneChip* offers no compiler support to partition source code.

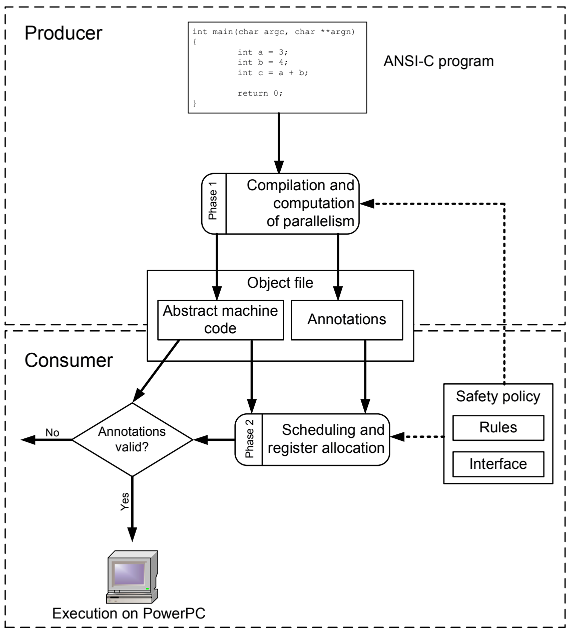
Spyder [21] is a superscalar processor with three execution units, which offer fine-grained reconfiguration at the gate level. But it suffers from the typical disadvantages of many reconfigurable architectures: First, Spyder is realized as a coprocessor, which is loosely coupled to the host workstation. This causes a major bottleneck between the host processor and the reconfigurable logic. Second, programm code must be split into two parts, with one part running on the host processor and the other part on Spyder. This difficult task is actually left to the programmer, because there is no automatic compiler available. Third, the user must provide a hardware configuration for the functional units, which cannot be changed at run-time by dynamic reconfiguration.

Hence, we will aim for a more coarse-grained, tightly integrated solution, where interconnection structures between fixed components can be reconfigured with minimum run-time effort.

3. STRUCTURE OF SYSTEM

Figure 1 shows the structure of our *CALS* system according to the *PPC* approach. The compiler transforms a C program into abstract machine code and appends code annotations. Then code and annotations are sent to the target machine in one object file. At load-time the scheduler arranges the abstract code for a concrete target machine with the help of the annotations. Apart from the partitioning in two phases, our system features another important property of the *PPC* approach: The *load-time scheduler* can operate very fast (linear time), while the compiler must determine the annotations with high computing effort.

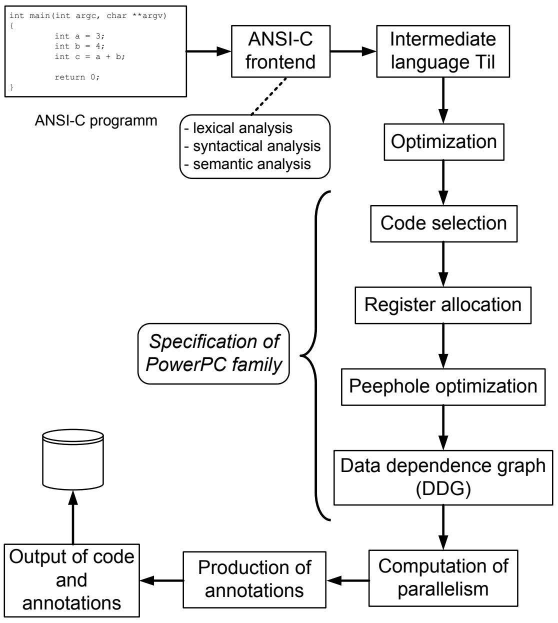
Actually, our system allows only simple validation of the annotations: The abstract machine code is rejected, if the size of the annotations is wrong. We plan to improve *CALS* to completely support the *PPC* approach by adding checks which guarantee that the annotations match the code.



```
int main(char argc, char **argn)
{
        int a = 3;
        int b = 4;
        int c = a + b;

        return 0;
}
```

Figure 1: Structure of the whole system


ANSI-C programm
ANSI-C frontend
Intermediate language Til
- lexical analysis
- syntactical analysis
- semantic analysis
Optimization
Code selection
Register allocation
Specification of PowerPC family
Peephole optimization
Data dependence graph (DDG)
Output of code and annotations
Production of annotations
Computation of parallelism

```
int main(int argc, char **argv)
{
        int a = 3;
        int b = 4;
        int c = a + b;

        return 0;
}
```

Figure 2: Structure of compiler

3.1 Compiler

The *load-time scheduler* must be as fast as possible. Therefore we have chosen scheduling at basic block level and skipped enhanced techniques such as *software pipelining* [23] or *trace scheduling* [11]. Furthermore, the evaluation in [28] has shown that *list scheduling* already achieves a significant speed-up over simple compilation without scheduling. For the *PowerPC* processors, the results of software pipelining fall short of the expectations, because these superscalar processors have far fewer execution units than *VLIW* machines. More recent *PowerPC* processors feature even fewer functional units, but compensate this with higher clock frequency and larger caches. Additionally, the *PowerPC 604* has a bottleneck in the write-back stage of the common instruction pipeline.

Although most basic blocks are rather short, loop unrolling can lead to much longer basic blocks, which are responsible for a significant share of a program's execution time. Hence even separate scheduling of these basic blocks can improve performance dramatically.

Conventional *VLIW* compilers allocate registers immediately after code scheduling. As the *load-time scheduler* must be fast, register allocation could be performed completely at compile-time. But this would overly constrain the scheduling phase at load-time. Hence we decided to split the register allocation between compile-time and load-time of a program: At compile-time all virtual registers, whose life spans include more than one basic block, are replaced by real registers. The other virtual registers are allocated during code scheduling at load-time. By these means, the possibilities for arranging instructions are scarcely limited.

As a result of these decisions we have developed an alternative compiler structure (see figure 2): First, registers are allocated by graph coloring [5, 3] according to the above description. Second, a peephole optimization is performed to remove superfluous instructions from the annotated code. After construction of the data-dependence graph (*DDG*), the fine-grained parallelism is computed. This structure has the outstanding advantage, that modifications of the machine program by spill code or peephole optimization are already considered when constructing the *DDG*. Finally, the code annotations are produced and added along the abstract machine code to an object file.

3.2 Scheduler

The structure of the *load-time scheduler* is shown in figure 3. First, the scheduler reads the object file and identifies the sections containing the abstract machine code and the annotations. Second, all functions are determined and linked to annotations. It is possible that some functions are not annotated, because linkers add code from static libraries or startup code, which is not annotated.

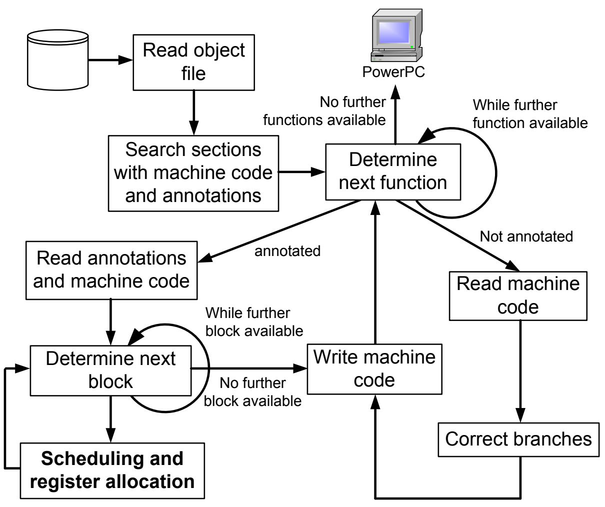

Read object file
PowerPC
No further functions available
While further function available
Search sections with machine code and annotations
Determine next function
annotated
Not annotated
Read annotations and machine code
Read machine code
While further block available
Determine next block
No further block available
Write machine code
Scheduling and register allocation
Correct branches

Figure 3: Structure of scheduler

Then all functions in the object file are processed in order: If a function is annotated, the associated annotations are read and scheduling is started. For each basic block, the scheduler invokes an algorithm, which arranges the machine instructions and performs register allocation. Finally, the code for the concrete target machine is written out.

Our approach supports partially annotated object files and linking against dynamic libraries with or without annotations. However non-annotated functions in the main executable cannot simply be copied, because the integrated register allocation may produce spill code. Hence, targets of branch instructions and function calls must be corrected in this case.

4. PREPARATION OF SCHEDULING

As our *load-time scheduler* operates at basic-block level, we can re-use some ideas of the *list scheduling* algorithm. List scheduling manages *DDG* nodes without predecessors in a *ready list*. Selection of nodes is mostly done by using heuristics such as earliest scheduling time or height in *DDG*. Our compiler determines *cumulative costs* [15] for all instructions, that influence scheduling at load-time. The cumulative costs for a node i are computed by taking the maximum cumulative costs of its successors in the *DDG* and adding the costs of executing node i itself. The costs of node i are modeled as the latency of the associated instruction. Hence the priority of an instruction is equivalent to the cumulative costs of its associated *DDG* node.

The cumulative costs can easily be computed by a recursive algorithm over the *DDG*. Then all instructions in each basic block are arranged by a topological sort of the *DDG*, which uses the cumulative costs as a secondary criterion. Such an ordering leads to the abstract machine code of one basic block. Finally, the code annotations are produced using the dependencies modelled by the *DDG* and the cumulative costs of each node.

4.1 Register Allocation

Our *CALS* approach splits register allocation between compile-time and load-time of a program. The allocation time of a virtual register can be determined by its life span. Additionally, the effects of virtual registers with short life spans on the allocation heuristic must be considered: Let us imagine a function with an exceptionally long basic block b and a virtual register v, whose life span covers the majority of the block and is completely included (see figure 4). We assume that the life spans of all other virtual registers cover more than one basic block. But this property does not imply that such life spans have a higher priority than the life span of v in terms of the allocation heuristic.

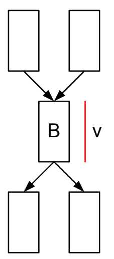


Figure 4: Important life span in a single basic block

Hence, it could happen that no real register is available for v at load-time, when using a strategy which just ignores v at compile-time. If the life span of v overlaps with many life spans of real registers, expensive spill code must be added. Therefore virtual registers such as v are already considered at compile-time without allocating a real register. By these means, a real register is *reserved* to make it likely that v gets a real register at load-time, if it would have got one at compile-time.

For preparation of register allocation at load-time, all life spans must be encoded in the annotations to avoid conflicts with already allocated registers. Additionally, the uses of virtual registers as operands of instructions must be remembered. For simplification, all special registers such as condition register fields are allocated at compile-time and hence are not considered at load-time.

4.2 Alternative Compiler Structure

The *CALS* compiler has an alternative structure, because effects of register allocation and peephole optimization can already be considered when constructing the *DDG*. Thus life spans must be stored temporarily after register allocation until the annotations have been generated. Each life span is modelled by its first and last instruction to avoid problems due to insertion of spill code. Additionally, the representation is independent of peephole optimization, unless the first or last instruction is removed. In such cases, affected life spans must be updated by considering the rules of the peephole optimization in table 1.

Rule	Original code	New code
(a) Superfluous move instructions	`move x, x`	
(b) Superfluous pairs of move instructions	`move x, y` `move y, x`	`move x, y`
(c) Superfluous load or store instructions	`store x, mem` `load x, mem`	`store x, mem`
(d) Superfluous branches	`branch label` `label:`	`label:`

Table 1: Rules of peephole optimization

Rule (d) can be neglected, because no register and hence no life span is concerned. In case (a), two life spans touch at a superfluous move instruction. Effects of removing this instruction can be avoided by a priori fusion of neighbouring life spans that share the same real registers. The same works for the case (c). Case (b) is more complicated, because life spans must be corrected by picking adjacent instructions as replacements (see figure 5): Life spans are always shortened, i.e. first and last instructions are replaced by their successor resp. predecessor. By these means, life spans could be shortened until they are empty and thus can be eliminated.

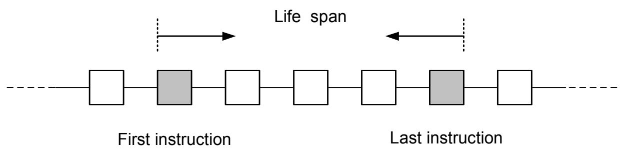


Figure 5: Replacement of first and last instruction of a life span

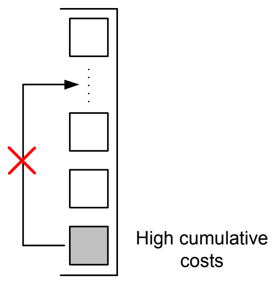


Figure 6: Situation demanding adjustment of cumulative costs

The above method guarantees that first and last instructions of all life spans are well-defined when encoding annotations. But the topological sort according to cumulative costs can modify the ordering of instructions for a life span (see figure 6): The last instruction has higher cumulative costs than some preceeding instructions and does not depend on them. Such cases could cause conflicts during register allocation at load-time, because life spans are irreversibly shortened. Therefore we decided to adjust the cumulative costs of all last instructions if necessary. In fact, cumulative costs are set to the minimum of the costs values of all other instructions for a certain life span. Additionally, the topological sort has been modified to prefer the earlier of two independent instructions with same cumulative costs.

4.3 Encoding of Life Spans

Life spans with real registers can be represented by three parameters:

- **Offset:** number of instructions between start of function and first instruction in life span
- **Length:** number of instructions in life span
- **Register:** number of real register (r0 = 0, ..., r31 = 31, f0 = 32, ..., f31 = 63)

Life spans with virtual registers are limited to one basic block by construction. Hence they can be partitioned according to the basic blocks of a function and encoded separately. They also are represented as an offset and length as well as register information and a list of forbidden registers. In this context the offset gives the distance from the start of the basic block to the first instruction.

The register information (see figure 7) contains information about the register type (integer or floating-point) and class (volatile or non-volatile) and a hint for a real register. The hint is used to speed up register allocation by the *load-time scheduler*: As motivated in 4.1, register allocation at compile-time considers all life spans. Hence real registers are known even for life spans, which are allocated at load-time. These allocations serve as hints that can be overridden by the *load-time scheduler*.

The register class indicates the preferred class to be used at load-time if possible. The use of a non-volatile register implies two additional instructions for saving and restoring the register's content at the beginning and end of a function. A volatile register is mostly used for function parameters. If the content needs to survive a function call, it must be saved before a call and restored before the next use. Hence, a volatile register should only be used if the number of function calls in the life span does not exceed 1.

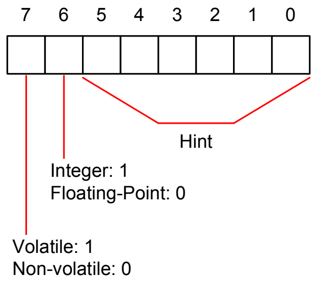


Figure 7: Register information for a virtual register

Conventional *PowerPC* compilers can determine the register class by counting the function calls inside all life spans. Although scheduling is done at load-time, we can estimate the register class by a heuristic method at compile-time (see figure 8): If there exists a path in the *DDG* from a function call C_1 to the first instruction in a life span resp. from the last instruction to a function call C_2, the calls C_1, C_2 will remain outside the life span after scheduling. If there is no such path for a function call C_{max}, this call *can* appear inside the life span after scheduling. If there are paths from the first instruction to a function call C_{min} and from C_{min} to the last instruction, this call will *always* end up inside the life span. Our compiler requests a volatile register, if the maximum number of calls (C_{max}, C_{min}) in the life span does not exceed 1.

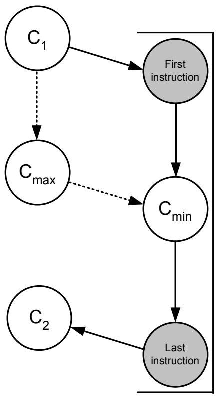


Figure 8: Function calls in *DDG*

The very efficient one-pass scheduling algorithm depends on advance information about forbidden registers. Otherwise dead locks can occur during register allocation at load-time. Hence, forbidden registers will never be assigned to a life span by our *load-time scheduler*. Additionally, the compiler never emits a hint that refers to a forbidden register in the first place.

Forbidden registers can be determined in a way similar to register classes: Our compiler checks the *DDG*, wether life spans of a given virtual and a given real register will always be disjoint after scheduling. If an overlap is possible, the real register is marked as forbidden for the other life span.

4.4 Encoding of Virtual Registers

Beyond the life spans proper, the *load-time scheduler* must know the uses of each virtual register to replace it with a real register. Representations of life spans mainly consist of an offset and a length, both measured in instructions. In the same manner, the uses of virtual registers can be encoded by counting register operands in a basic block from top to bottom and left to right within each instruction: The first use of a virtual register is modelled as the offset from the beginning of a basic block. All following uses are represented as their distance from the previous use. A naive encoding would just store the number of uses and the offset values into the annotations.

To reduce the size of annotations, we developed an efficient encoding, which reuses the space of the unused register operands themselves: Every operand for an integer or floating-point register of a *PowerPC* processor is 5 bits wide. Hence a value, that can be encoded in 5 bits, is stored in place of the register operand. Otherwise, the value is stored in the annotations and the register operand is marked accordingly. Only the number of uses and the first offset value are written to the annotations by default. By these means, only two values end up in the annotations per life span in the best case.

4.5 Encoding of Annotations

An ideal encoding of the annotations should be both compact and flexible. Therefore we used the *VLQ* encoding (Variable Length Quantity) known from the music file format *MIDI* [1]. *VLQ* encoding and decoding requires less time and memory than other techniques such as Huffman [19] or Lempel-Ziv [31].

The *VLQ* encoding of a number consists of a sequence of bytes, where data is stored in the last seven bits (right-justified) of a byte. The last byte of a sequence is marked by setting the MSB to 0. In all predecing bytes the MSB is set to 1. Hence values between 0 and 127 can be represented in one byte using *VLQ*. Because most values are even smaller than 16 in our context, we use nibbles instead of bytes as basic units of our *VLQ* encoding.

5. LOAD-TIME SCHEDULER

Here we present the structure of the core *load-time scheduler* (see figure 9, bold-printed in figure 3). It consists of the scheduling algorithm and three managers.

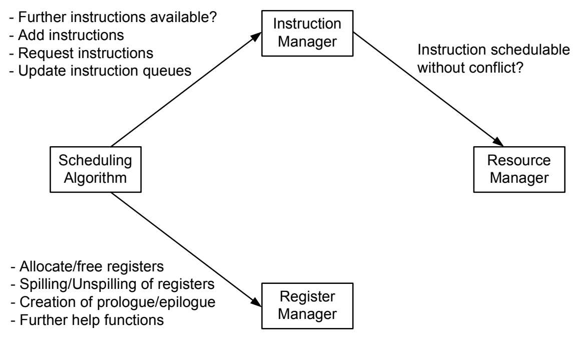

- Further instructions available?
- Add instructions
- Request instructions
- Update instruction queues
Instruction Manager
Instruction schedulable without conflict?
Scheduling Algorithm
Resource Manager
- Allocate/free registers
- Spilling/Unspilling of registers
- Creation of prologue/epilogue
- Further help functions
Register Manager

Figure 9: Structure of *load-time scheduler*

The *instruction manager* organizes the instructions of a basic block and has a structure as shown in figure 10. The algorithm submits all instructions of a basic block to the *instruction manager* before scheduling and requests instructions one by one during scheduling. For efficiency, instructions are managed in several queues: A single queue contains all instructions with at least one unscheduled predecessor. Other instructions are distributed to special queues, which are sorted according to decreasing cumulative costs.

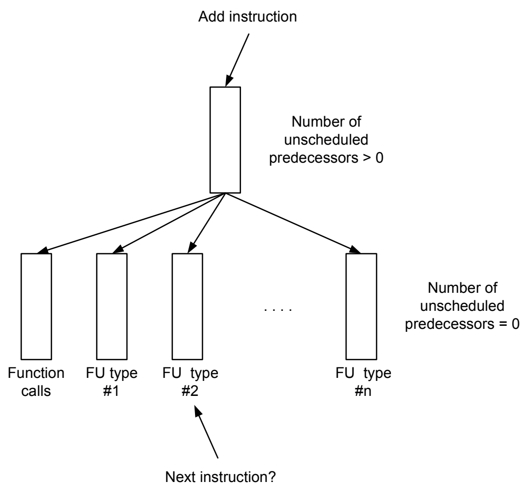


Figure 10: Queues of the *instruction manager*

Special queues correspond to the functional unit types of the concrete target machine. In addition there is a queue for function calls, because these are typically not executed on a separate functional unit, but together with branch instructions on a branch unit. By the means of a separate queue, function calls can be preferred over other instructions to reduce code size and hence execution time. Otherwise many instructions would be scheduled prematurely before function calls. This would imply additional code for saving and restoring volatile registers.

New instructions are inserted into the appropriate queues. After handling each cycle, the scheduler propagates all instructions without unscheduled predecessors to the correct special queue (l. 31 of the scheduling algorithm).

The *resource manager* organizes all resources of the concrete target machine and interacts with the other components as follows: During each cycle, the algorithm requests instructions for each functional unit from the *instruction manager* (ll. 6/8 of the scheduling algorithm). It searches through the associated special queue for the first schedulable instruction. These schedulability checks are performed by the mentioned *resource manager*. If an appropriate instruction is found, it is returned to the algorithm. Otherwise, the scheduler cannot utilize the functional unit during this cycle. In contrast to conventional techniques such as *list scheduling*, the number of instructions, which are considered per cycle, is limited (see 5.1).

The *register manager* organizes all registers of the concrete target machine. The algorithm can (de-)allocate registers and select registers for spilling. Additionally, the *register manager* creates machine code for prologue and epilogue parts of functions.

5.1 Scheduling Algorithm

Algorithm 5.1 shows the pseudo code of our scheduling algorithm. For simplification, we neglect the integrated register allocation (l. 13) as well as spilling. The algorithm primarily comprises a loop (ll. 2-32), which is executed until all instructions have been scheduled. The loop consists of three parts: First, new instructions for a cycle are requested from the *instruction manager* and marked as executing (ll. 3-20). Then the cycle number is incremented by 1 (l. 21). Finally, all completed instructions are finished and the number of unscheduled predecessors is decremented at their immediate successors (ll. 22-30).

```
Require: Instructions of basic block with cumulative costs
    and dependencies
Ensure: Scheduled instructions as list

 1: $cycle := 0; exe\_list := \varnothing; block\_list := \varnothing;$
 2: while $not\ iman\_empty()$ do
 3:     for all $fu \in FU$ do
 4:         $real\_fu := real\_funcunit[fu];$
 5:         if $fu = calls$ then
 6:             $instr := iman\_next\_call();$
 7:         else
 8:             $instr := iman\_next\_instr(real\_fu);$
 9:         end if
10:         if $instr = \varnothing$ then
11:             $continue;$
12:         end if
13:         $update\_registers(instr);$
14:         if $peephole(instr)$ then
15:             $continue;$
16:         end if
17:         $endcycle[instr] := cycle + latency[instr];$
18:         $exe\_list := insert(exe\_list, instr);$
19:         $block\_list := append(block\_list, instr);$
20:     end for
21:     $cycle := cycle + 1;$
22:     while $exe\_list \neq \varnothing$ do
23:         $instr := first(exe\_list);$
24:         if $endcycle[instr] > cycle$ then
25:             for all $succ \in succs[instr]$ do
26:                 $num\_precs[succ] := num\_precs[succ] - 1;$
27:             end for
28:         end if
29:         $exe\_list := rest(exe\_list);$
30:     end while
31:     $iman\_update();$
32: end while
```

Algorithm 5.1: Algorithm of *load-time scheduler*

The number of instructions, which can be considered by the *instruction manager* per cycle, is limited by $\lambda = 2 \times \#FUs$. We decided against bounding the number per queue, because evaluation showed that the queues clearly vary in length. With our bound, the *instruction manager* can handle more instructions of a long queue, if other queues are very short or even empty.

Hence our algorithm needs linear time only: Given a basic block with n instructions. We assume initially that all instructions have uniform latency l. As the maximum number of instructions considered is limited by λ, our algorithm needs only constant effort λ for scheduling one cycle. In worst case, the algorithm covers $n \cdot l$ cycles and we get a total runtime of $O(n \cdot l \cdot \lambda) = O(n)$. The *list scheduling* algorithm does not place an upper bound on the number of instructions considered and thus has quadratic complexity.

Now, we permit that latencies of instructions are not uniform. Again, our algorithm needs only linear time, because its effort per cycle is still constant. The *list scheduling* algorithm has quadratic or higher complexity in this context.

5.2 Strategies for Scheduling

The strategy of our scheduler prefers instructions with high cumulative costs. But in situations with only a few free registers left much spill code would be added, because the register needs of instructions are ignored completely.

Hence we decided to support two scheduling strategies, which are selected depending on the number of free registers: The first strategy is based on cumulative costs as outlined above. The second strategy uses the number of life spans that end at an instruction as its primary criterion. In case of equality, the instruction with higher cumulative costs is preferred. By selecting an instruction which finishes many life spans, the number of free registers is increased and hence spilling is avoided. On the other hand, instructions with high cumulative costs are possibly discriminated.

Selection and application of strategies is integrated in the *instruction manager*. Hence the scheduling algorithm is completely independent of the active strategy and new strategies can be added easily. During a strategy change, all special queues of the *instruction manager* are re-arranged according to the new sort order. This operation is quite expensive compared to the common algorithm and should be performed as seldom as possible.

Therefore, we use a hysteresis: If the number of integer or floating-point registers falls below 4, the second strategy is chosen. The value 4 should be high enough to respond to a high register need in a timely manner. A return to the first strategy occurs only after at least 8 free registers of each type are available.

6. EVALUATION

Our current prototype of the *CALS* system does not forward the scheduled machine code to the loader directly. Instead, it updates the object code in memory and writes back an object file which is then executed by the *PowerPC* simulator *PSIM*. The simulator allows cycle-accurate execution, i.e. the evaluation results are not influenced by dynamic effects like caching etc. It supports the *PowerPC* processors *601*, *603*, *603e* and *604*. Additionally, a variant of the 604 with only one *SCIU* (Single-Cycle Integer Unit) can be simulated. In the following, we call this variant *750CX*, because both machines have a great deal in common, neglecting unimportant technical details.

6.1 Modelling of PowerPC processors

We decided to evaluate our *CALS* approach for the processors *604*, *603* and *750CX*. The *603e* was not modelled, because it is almost similar to the *603*. Just as well, the *601* is very constrained in our context due to poor degree of parallelism and a common cache for instructions and data.

We adopted a machine model for the *604* from a project presented in [28]. The model was specified with our machine specification language *UPSLA* (Unified Processor Specification LAnguage), which represents the successor of the language *Masl* used for [28]. The *UPSLA* compiler generates machine specific parts of the compiler backend and the *load-time scheduler*. Additionally, we developed a second specification language and a compiler to generate an instruction decoder as well as operand access functions for the scheduler. We plan to merge both languages in the future.

In contrast to *Masl*, a specification in *UPSLA* consists of a sequence of element definitions. The element types are fixed and include instructions, instruction attributes, resources, functional units and register banks. Mappings between resources, functional units and register banks can be modelled with so-called mapped resources. Processor specification is simplified by features of object-oriented languages like element inheritance, abstract elements, grouping of elements and equivalence classes with identical resource mappings.

The *750CX* could be modelled easily by removing the second *SCIU* from the specification of the *604*. A *603* has only one IU (Integer Unit), while a *604* features two *SCIUs* and one *MCIU* (Multiple-Cycle Integer Unit). For simplification, we modelled the *IU* by a separate *SCIU* and *MCIU* with a virtual resource for mutual exclusion. Additionally, the latencies of some multiplication and division instructions had to be modified.

6.2 Benchmarks

For the *initial* evaluation we have chosen six rather small programs from the Stanford Benchmarks, which have been made conformant to the *ANSI C* standard:

- *b_bubble*: bubble sort
- *b_mm*: matrix multiplication (floating-point entries)
- *b_quick*: quick sort
- *b_trees*: tree sort
- *intmm*: matrix multiplication (integer entries)
- *puzzle*: compute bound program

In the following sections, we discuss the evaluation of *CALS* (Code Annotations for Load-time Scheduling).

6.3 Runtime

Here we compare *CALS* to simple compilation without scheduling and conventional techniques such as *ASAP* and *ALAP*. Figure 11 shows a summary of the runtime improvements for all machines. First, our approach achieves promising results of 8.4% in average over simple compilation. The highest improvement is obtained for *intmm* on a *604* (23.8%). The worst case occurs for *b_mm* on a *603* (0.2%), but still represents a minimal speed-up.

Second, runtime is only 1.7% higher in average than for *ASAP*. In comparison with *ALAP*, we can even observe an average speed-up of 1.8%. Only the *b_trees* benchmark shows a degradation in performance and is responsible for nearly all of the worst case figures.

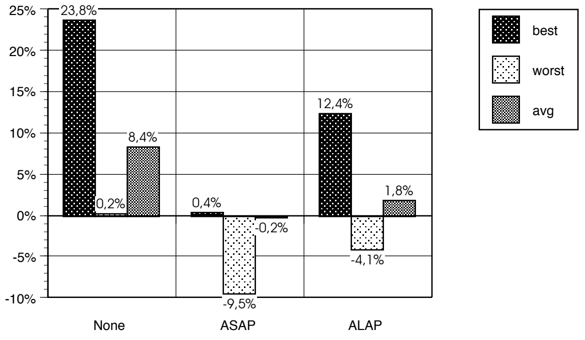


Figure 11: Comparison with conventional techniques

Now we consider the results for certain target machines (figure 12-14): The *604* offers the highest parallelism of the three machines and also shows the largest speed-up over simple compilation. But the best cases compared to *ASAP* and *ALAP* are achieved on a *750CX*, which has only one *SCIU*. Our *load-time scheduler*, in contrast to conventional schedulers, takes spill code generated at compile-time into account. Precise placement of spill code is more important on the *750CX* with is limited dispatching abilities.

To summarize, our approach achieves noticable improvements over simple compilation without scheduling. Additionally, *CALS* obtains results comparable to list scheduling or even outperforms it.

As our *load-time scheduler* knows the concrete target machine precisely, *CALS* should obtain shorter runtimes than static *list scheduling* for the wrong target machine. Otherwise, the additional effort at load-time could not be justified and one could just perform *list scheduling* for a generic processor of the *PowerPC* family. Again, the evaluation revealed the feasibility of our approach: *CALS* achieves performance improvements of up to 12.4% over static *list scheduling* for the wrong target machine.

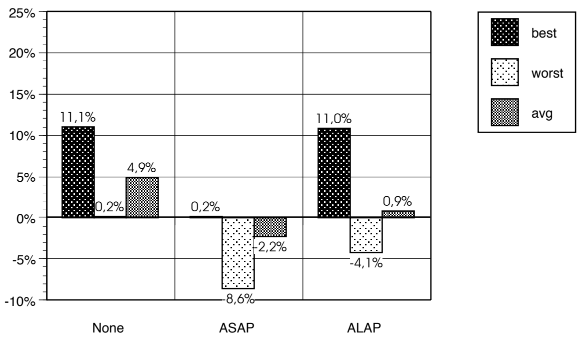


Figure 12: Comparison with conventional techniques on *PPC 603*

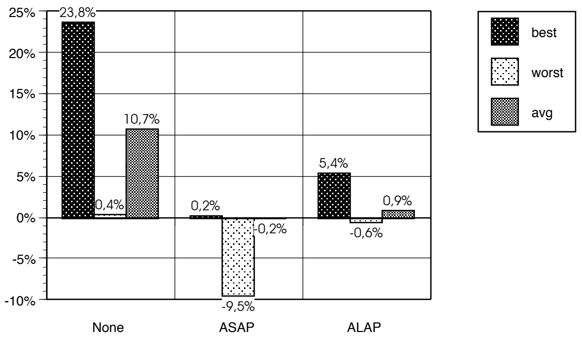


Figure 13: Comparison with conventional techniques on *PPC 604*

The best results can be observed on the *750CX*, which has medium parallelism of the three machines. At first glance, this is surprising, because the *603* has the lowest parallelism. Hence, code that was scheduled for a better machine should exhibit a massive slow-down on the *603*. But at the beginning we mentioned that the best results compared to *ASAP* and *ALAP* for the correct target machine are achieved on

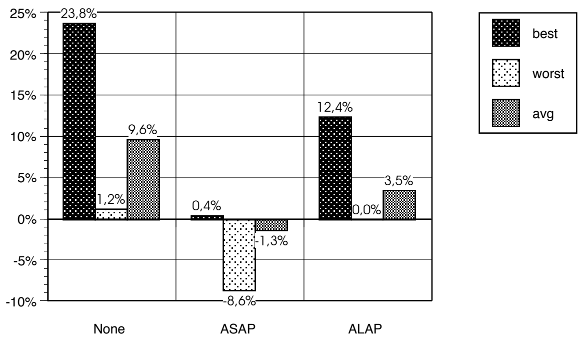


Figure 14: Comparison with conventional techniques on *PPC 750CX*

a *750CX*. Hence, the speed-up by *CALS* on a *750CX* more than compensates the slowdown due to the low parallelism of the *603*.

On the *604*, the improvements over static *list scheduling* for the wrong target machine are minimal. This is to be expected, because the *604* offers the highest parallelism and therefore the *dispatcher* has the biggest potential for re-arrangement of poorly scheduled code.

Finally, we can conclude that the promising results of our approach justify the additional overhead at load-time. Additionally, the effort for load-time scheduling is low, because our algorithm only needs linear time due to the annotations.

6.4 Size of Annotations

Here we discuss the relative size of the annotations with respect to the code size. The object files mainly consist of non-annotated functions from libraries, which were added during linking. Hence, the relative size with respect to the total code size is very small (2-4% with all optimizations). One exception is *puzzle* with a relative annotation size of 11%, because large offset values (up to 500) of life spans with real registers must be encoded. Thus, future work should also review the computation of values to be encoded, next to improving the encoding method.

But the total code size is not really representative, because library functions could also be annotated in future. If only the size of annotated code is considered, the relative size of annotations is 157-220% (avg. 180%) when using the original *VLQ* method and a naive encoding of virtual register uses. The usage of nibbles instead of bytes as basic units of *VLQ* has reduced annotation size by 24-29%. Additional savings of 6-8% have been achieved by encoding the uses of virtual registers in the space of the register operands. Currently, the relative size is 106-154% (avg. 125%), which represents an improvement of 30-33%.

In the future, we will evaluate other encodings such as Huffman or Lempel-Ziv, although more time and memory is required compared to *VLQ*. Additionally, we will reconsider the exact information stored in the annotations.

6.5 Design-specific Aspects

In this paper, we have presented several ideas to support efficient scheduling like split register allocation, computation of hints and forbidden registers as well as preference of function calls. Our evaluation has shown that these concepts have met or exceeded our expectation in most cases. Hence, we do not discuss runtime results in detail at this point. Instead, we focus shortly on design-specific aspects:

The encoding of a life span with virtual register contains the preferred register class and a hint for a real register. First, our evaluation has shown that volatile registers are preferred and also used at load-time in most cases. Second, our compiler can determine 40% more hints, if forbidden registers are computed by using the *DDG*. A naive computation would just forbid all real registers which are used in a basic block. But such a strategy would partition real registers according to compile-time and load-time. Hence, register need would increase and many in fact superfluous instructions could not be eliminated by the peephole optimization at load-time. Third, the scheduler can re-use 50-80% of the hints at load-time, which implies very fast register allocation without additional effort. Hence, the register allocation at load-time differs from register allocation at compile-time only slightly (hints), but at the same time enlarges the freedom of our *load-time scheduler*.

Finally, we focus on the special queues of the *instruction manager*. The special queues for function calls, *BPU*, and *FPU* resp. *MCIU* and *LSU* contain at most one or two instructions. Only the *SCIU* queue contains up to 14 instruction. Therefore bounding the number of instructions considered per cycle globally is better than a separate upper limit per queue. Additionally, evaluation has shown that complete register allocation at compile-time severely constrains the scope of the *load-time scheduler*: Maximum lengths of special queues decrease to 1/2 (or even 1/7 for *puzzle*) of the observed values.

7. CONCLUSION

We have implemented a novel system, which prepares parallelization and register allocation at compile-time and performs scheduling at load-time of a program. Our *CALS* approach achieves noticable improvements over simple compilation without scheduling. Its results are comparable to list scheduling or even better by up to 12.4%. Additionally, the effort for load-time scheduling is low, because our algorithm only needs linear time due to its reliance on the precomputed annotations. At a modest cost in file size, our approach provides higher performance for mobile code in a heterogenous network, where the precise machine model is not known at compile-time.

Our next step will be the evaluation of *CALS* based on larger, realistic application programs. Furthermore, we envision to extend our methodology to reconfigurable architectures, where the target machine adapts itself to the characteristics of a program, which in turn is scheduled by a *load-time scheduler*.

8. REFERENCES

[1] M. M. Association. The Complete MIDI 1.0 Detailed Specification. Technical report, MIDI Manufactures Association, 2001.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77, June 1999.

[3] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[4] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, 2000.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[6] T. M. Conte and S. W. Sathaye. Optimization of VLIW Compatibility Systems Employing Dynamic Rescheduling. *International Journal of Parallel Programming*, 25(2):83–112, 1997.

[7] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003.

[8] K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *ISCA*, pages 26–37, 1997.

[9] D. R. Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 160–170. ACM Press, 1996.

[10] D. R. Engler and T. A. Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–272. ACM Press, 1994.

[11] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comps., C-30 7/81*, pages 478–490, 1981.

[12] M. Franz and T. Kistler. Slim Binaries. *Commun. ACM*, 40(12):87–94, 1997.

[13] C. W. Fraser and D. R. Hanson. A Retargetable Compiler for ANSI C. Technical Report CS–TR–303–91, Princeton, N.J., 1991.

[14] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.

[15] J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 2nd international conference on Supercomputing*, pages 442–452. ACM Press, 1988.

[16] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera Reconfigurable Functional Unit. In K. L. Pocek and J. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96. IEEE Computer Society Press, 1997.

[17] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[18] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research 9*, pages 841–848, 1961.

[19] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. 40(9):1098–1101, 1952.

[20] IBM and Motorola. PowerPC 604 RISC Microprocessor User's Manual. Technical report, 1996.

[21] C. Iseli and E. Sanchez. Beyond Superscalar Using FPGAs. In *Proceedings of the International Conference on Computer Design*, Oct. 1993.

[22] A. Klaiber. The Technology Behind Crusoe Processors. Technical report, 2000.

[23] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328. ACM Press, 1988.

[24] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[25] M. D. Nemirovsky, F. Brewer, and R. C. Wood. DISC: Dynamic Instruction Stream Computer. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 163–171. ACM Press, 1991.

[26] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.*, 49(5):465–481, 2000.

[27] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary Translation. *Commun. ACM*, 36(2):69–81, 1993.

[28] E. Stümpel, M. Thies, and U. Kastens. VLIW Compilation Techniques for Superscalar Architectures. In K. Koskimies, editor, *Proceedings 7th International Conference on Compiler Construction CC '98*, number 1383 in Lecture Notes in Computer Science. Springer Verlag, März 1998.

[29] S. Weiss and J. E. Smith. *Power and the PowerPC*. Morgan Kaufmann Publishers, Inc., 1994.

[30] R. Wittig and P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[31] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.