This is the authors' final manuscript, accepted 2010. The final version appeared in *ACM Transactions in Embedded Computing Systems*, vol. 11S, number 1, Article 19, June 2012, 24 pages. DOI=10.1145/2180887.2180896 http://doi.acm.org/10.1145/2180887.2180896 ©ACM.

Integrated Code Generation for Loops

MATTIAS ERIKSSON and CHRISTOPH KESSLER
Linköping university

Code generation in a compiler is commonly divided into several phases: instruction selection, scheduling, register allocation, spill code generation, and, in the case of clustered architectures, cluster assignment. These phases are interdependent; for instance, a decision in the instruction selection phase affects how an operation can be scheduled. We examine the effect of this separation of phases on the quality of the generated code. To study this we have formulated optimal methods for code generation with integer linear programming; first for acyclic code and then we extend this method to modulo scheduling of loops. In our experiments we compare optimal modulo scheduling, where all phases are integrated, to modulo scheduling, where instruction selection and cluster assignment are done in a separate phase. The results show that, for an architecture with two clusters, the integrated method finds a better solution than the non-integrated method for 27% of the instances.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code Generation,Optimization*

General Terms: Algorithms, Experimentation, Performance, Theory

Additional Key Words and Phrases: Code generation, clustered VLIW architectures, modulo scheduling

1. INTRODUCTION

A processor in an embedded device often spends the major part of its life executing a few lines of code over and over again. Finding ways to optimize these lines of code before the device is brought to the market could make it possible to run the application on cheaper or more energy efficient hardware. This fact motivates spending large amounts of time on aggressive code optimization. In this paper we aim at improving current methods for code optimization by exploring ways to generate provably optimal code (in terms of throughput).

Code generation is performed in the *back end* of a compiler; in essence, it is the process of creating executable code from the previously generated intermediate

Author's addresses: Dept. of Computer and Information Science, Linköpings universitet, SE-581 83 Linköping, Sweden. Email: {mater,chrke}@ida.liu.se.
This work has been supported by The Swedish national graduate school in computer science (CUGS), Vetenskapsrådet (VR) and Stiftelsen för strategisk forskning (SSF).
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1539-9087/2012/06-ART19$10.00

representation (IR). One way to do this is to perform three phases in some sequence:

—*Instruction selection phase* — Select target instructions matching the IR. This phase includes resource allocation.
—*Instruction scheduling phase* — Map the selected instructions to time slots on which to execute them.
—*Register allocation phase* — Select registers in which intermediate values are to be stored.

While doing the phases in sequence is simpler and less computationally heavy, the phases are interdependent. Hence, integrating the phases of the code generator gives more opportunities for optimization. The cost of integrating the phases is that the size of the solution space increases: there is a combinatorial explosion when decisions in all phases are considered simultaneously. This is especially the case when we consider complicated processors with clustered register files and functional units where many different target instructions may be applied to a single IR operation, and with both explicit and implicit transfers between the register clusters.

In this paper we are interested in code generation for *very long instruction word* (VLIW) architectures [Fisher 1983]. For VLIW processors the issued long instruction words contain multiple operations that are executed in parallel. This means that all instruction level parallelism is static, i.e. the compiler (or assembler level programmer) decides which operations are to be executed at the same point in time. Our focus is particularly on *clustered* VLIW architectures in which the functional units of the processor are limited to using a subset of the available registers [Fernandes 1998]. The motivation behind clustered architectures is to reduce the number of register ports and thereby making the processor use less silicon and be more scalable. This clustering makes the job of the compiler even more difficult since there are now even stronger interdependencies between the phases of the code generation. For instance, which instruction (and thereby also functional unit) is selected for an operation influences to which register the computed value may be written.

For ease of presentation, we begin with an integer linear programming model for acyclic code generation, i.e. for basic blocks[1], in Section 2. This section also contains an experimental comparison of the integer linear programming model to a heuristic based on genetic algorithms. In Section 3 we extend the integer linear programming model to modulo scheduling. Additionally we show some theoretical properties of the algorithm and its search space and show results of an extensive experimental evaluation where we compare the fully integrated method to a method where instruction selection and cluster assignment is done in a separate phase. Section 4 lists related work in acyclic and cyclic integrated code generation and Section 5 concludes the paper.

2. INTEGRATED CODE GENERATION WITH INTEGER LINEAR PROGRAMMING

For optimal code generation for basic blocks we use an integer linear programming formulation. In this section we will introduce all parameters, variables and con-

[1]A *basic block* is a block of code that contains no jump instructions and no jump target other than the beginning of the block. I.e., when the flow of control enters the basic block all of the operations in the block are executed exactly once.

straints which are used by the CPLEX solver to generate a schedule with minimal execution time. This model integrates instruction selection (including cluster assignment), instruction scheduling and register allocation. An advantage of using integer linear programming is that a mathematically precise description is generated as a side effect. Also, the integer linear programming model is natural to extend to modulo scheduling, as we show in Section 3. The integer linear programming model presented here is based on a series of models previously published in [Bednarski and Kessler 2006; Eriksson et al. 2008; Eriksson and Kessler 2009].

2.1 Data flow graph

A basic block is modeled as a directed acyclic graph (DAG) $G = (V, E)$, where $E = E_1 \cup E_2 \cup E_m$. The set V contains intermediate representation (IR) nodes, the sets $E_1, E_2 \subset V \times V$ represent edges between operations and their first and second operand respectively. Dependences that are not true data dependences are modeled with the set $E_m \subset V \times V$. The integer parameter Op_i describes operators of the IR-nodes $i \in V$.

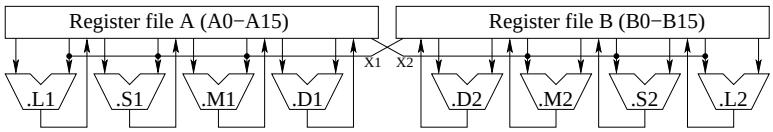
2.2 Instruction set

The instructions of the target machine are modeled by the set $P = P_1 \cup P_{2+} \cup P_0$ of patterns. P_1 is the set of *singletons*, which only cover one IR node. The set P_{2+} contain *composites*, which cover multiple IR nodes (used e.g. for multiply-and-add which covers a multiplication immediately followed by an addition). And the set P_0 consists of patterns for *non-issue* instructions which are needed when there are IR nodes in V that do not have to be covered by an instruction, e.g. an IR node representing a constant value that needs not be loaded into a register. The IR is low level enough so that all patterns model exactly one (or zero in the case of P_0) instructions of the target machine. When we use the term *pattern* we mean a pair consisting of one instruction and a set of IR-nodes that the instruction can implement. I.e., an instruction can be paired with different sets of IR-nodes and a set of IR-nodes can be paired with more than one instruction. For instance, on the TI-C62x DSP processor (see Figure 1) an addition can be done with any of twelve different instructions (not counting the multiply-and-add instructions): `ADD.L1`, `ADD.L2`, `ADD.S1`, `ADD.S2`, `ADD.D1`, `ADD.D2`, `ADD.L1X`, `ADD.L2X`, `ADD.S1X`, `ADD.S2X`, `ADD.D1X` or `ADD.D2X`.

For each pattern $p \in P_{2+} \cup P_1$ we have a set $B_p = \{1, \ldots, n_p\}$ of generic nodes for the pattern. For composites we have $n_p > 1$ and for singletons $n_p = 1$. For composite patterns $p \in P_{2+}$ we also have $EP_p \subset B_p \times B_p$, the set of edges between the generic pattern nodes. Each node $k \in B_p$ of the pattern $p \in P_{2+} \cup P_1$ has an associated operator number $OP_{p,k}$ which relates to operators of IR nodes. Also, each $p \in P$ has a *latency* L_p, meaning that if p is scheduled at time slot t the result of p is available at time slot $t + L_p$.

2.3 Resources and register sets

We model the resources of the target machine with the set $\mathcal{F}$ and the register banks with the set $\mathcal{RS}$. The binary parameter $U_{p,f,o}$ is 1 iff the instruction with pattern $p \in P$ uses the resource $f \in \mathcal{F}$ at time step o relative to the issue time. Note that this allows for multiblock [Kessler et al. 2007] and irregular reservation tables [Rau



1: The Texas Instruments TI-C62x processor has two register banks and 8 functional units [Texas Instruments Incorporated 2000]. The crosspaths X1 and X2 are modeled as resources, too.

1994]. $\mathcal{R}_r$ is a parameter describing the number of registers in the register bank $r \in \mathcal{RS}$. The *issue width* is modeled by ω, i.e. the maximum number of instructions that may be issued at any time slot.

For modeling transfers between register banks we do not use regular instructions (note that transfers, like spill instructions, do not cover nodes in the DAG). Instead we let the integer parameter $LX_{r,s}$ denote the latency of a transfer from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$. If no such transfer instruction exists we set $LX_{r,s} = \infty$. And for resource usage, the binary parameter $UX_{r,s,f}$ is 1 iff a transfer from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$ uses resource $f \in \mathcal{F}$. Note that we can also integrate spilling into the formulation by adding a virtual register file to $\mathcal{RS}$ corresponding to the memory, and then have transfer instructions to and from this register file corresponding to stores and loads. See Figure 1 for an illustration of a clustered architecture.

Lastly, we have the sets $PD_r, PS1_r, PS2_r \subset P$ which, for all $r \in \mathcal{RS}$, contain the pattern $p \in P$ iff p stores its result in r, takes its first operand from r or takes its second operand from r, respectively.

2.4 Solution variables

The parameter $t_{\max}$ gives the last time slot on which an instruction may be scheduled. We also define the set $T = \{0, 1, 2, \ldots, t_{\max}\}$, i.e. the set of time slots on which an instruction may be scheduled. For the acyclic case $t_{\max}$ is incremented until a solution is found.

We have the following binary solution variables:

—$c_{i,p,k,t}$, which is 1 iff IR node $i \in V$ is covered by $k \in B_p$, where $p \in P$, issued at time $t \in T$.

—$w_{i,j,p,t,k,l}$, which is 1 iff the DAG edge $(i, j) \in E_1 \cup E_2$ is covered at time $t \in T$ by the pattern edge $(k, l) \in EP_p$ where $p \in P_{2^+}$ is a composite pattern.

—$s_{p,t}$, which is 1 iff the instruction with pattern $p \in P_{2^+}$ is issued at time $t \in T$.

—$x_{i,r,s,t}$, which is 1 iff the result from IR node $i \in V$ is transfered from $r \in \mathcal{RS}$ to $s \in \mathcal{RS}$ at time $t \in T$.

—$r_{rr,i,t}$, which is 1 iff the value corresponding to the IR node $i \in V$ is available in register bank $rr \in \mathcal{RS}$ at time slot $t \in T$.

We also have the following integer solution variable:

—τ is the first clock cycle on which all latencies of executed instructions have expired.

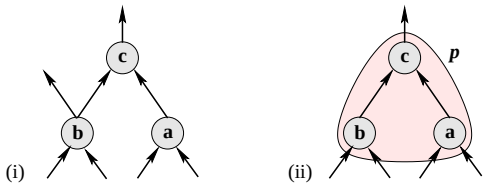


Fig. 2. (i) Pattern p can not cover the set of nodes since there is another outgoing edge from b, (ii) p covers nodes a, b, c.

2.5 Removing impossible schedule slots

We can significantly reduce the number of variables in the model by performing soonest-latest analysis on the nodes of the graph. Let $L_{\min}(i)$ be 0 if the node $i \in V$ may be covered by a composite pattern, and the lowest latency of any instruction $p \in P_1$ that may cover the node $i \in V$ otherwise. Let $\text{pre}(i) = \{j : (j,i) \in E\}$ and $\text{succ}(i) = \{j : (i,j) \in E\}$. We can recursively calculate the soonest and latest time slot on which node i may be scheduled:

$$soonest'(i) = \begin{cases} 0 & \text{, if } |\text{pre}(i)| = 0 \\ \max_{j \in \text{pre}(i)}\{soonest'(j) + L_{\min}(j)\} & \text{, otherwise} \end{cases} \tag{1}$$

$$latest'(i) = \begin{cases} t_{\max} & \text{, if } |\text{succ}(i)| = 0 \\ \max_{j \in \text{succ}(i)}\{latest'(j) - L_{\min}(i)\} & \text{, otherwise} \end{cases} \tag{2}$$

$$T_i = \{soonest'(i), \ldots, latest'(i)\} \tag{3}$$

We can also remove all the variables in c where no node in the pattern $p \in P$ has an operator number matching i. We can view the matrix c of variables as a sparse matrix; the constraints dealing with c must be written to take this into account. In the following mathematical presentation $c_{i,p,k,t}$ is taken to be 0 if $t \notin T_i$ for simplicity of presentation.

2.6 Optimization constraints

2.6.1 *Optimization objective.* The objective of the integer linear program is to minimize the execution time:

$$\min \tau \tag{4}$$

The execution time is the latest time slot where any instruction terminates. For efficiency we only need to check for execution times for instructions covering an IR node with out-degree 0, let $V_{\text{root}} = \{i \in V : \nexists j \in V,\ (i,j) \in E\}$:

$$\forall i \in V_{\text{root}},\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(t + L_p) \leq \tau \tag{5}$$

2.6.2 *Node and edge covering.* Exactly one instruction must cover each IR node:

$$\forall i \in V, \quad \sum_{\substack{p \in P \\ k \in B_p \\ t \in T}} c_{i,p,k,t} = 1 \tag{6}$$

Equation 7 sets $s_{p,t} = 1$ iff the composite pattern $p \in P_{2^+}$ is used at time $t \in T$. This equation also guarantees that either all or none of the generic nodes $k \in B_p$ are used at a time slot:

$$\forall p \in P_{2^+},\ \forall t \in T,\ \forall k \in B_p, \quad \sum_{i \in V} c_{i,p,k,t} = s_{p,t} \tag{7}$$

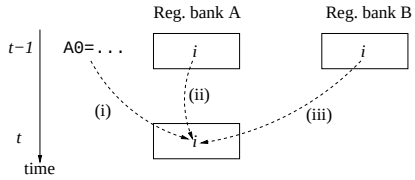


Fig. 3. A value may be live in a register bank A if: (i) it was put there by an instruction, (ii) it was live in register bank A at the previous time step, and (iii) the value was transferred there by an explicit transfer instruction.

An edge within a composite pattern may only be active if there is a corresponding edge (i, j) in the DAG and both i and j are covered by the pattern, see Figure 2:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall p \in P_{2^+},\ \forall t \in T,\ \forall (k,l) \in EP_p,$$
$$2w_{i,j,p,t,k,l} \leq c_{i,p,k,t} + c_{j,p,l,t} \tag{8}$$

If a generic pattern node covers an IR node, the generic pattern node and the IR node must have the same operator number:

$$\forall i \in V,\ \forall p \in P,\ \forall k \in B_p,\ \forall t \in T, \quad c_{i,p,k,t}(Op_i - OP_{p,k}) = 0 \tag{9}$$

2.6.3 *Register values.* A value may only be present in a register bank if: it was just put there by an instruction, it was available there in the previous time step, or just transfered to there from another register bank (see visualization in Figure 3):

$$\forall rr \in \mathcal{RS},\ \forall i \in V,\ \forall t \in T,$$
$$r_{rr,i,t} \leq \sum_{\substack{p \in PD_{rr} \cap P \\ k \in B_p}} c_{i,p,k,t-L_p} + r_{rr,i,t-1} + \sum_{rs \in \mathcal{RS}} (x_{i,rs,rr,t-LX_{rs,rr}}) \tag{10}$$

The operand to an instruction must be available in the correct register bank when we use it. A limitation of this formulation is that composite patterns must have all operands and results in the same register bank:

$$\forall (i,j) \in E_1 \cup E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS},$$
$$r_{rr,i,t} \geq \sum_{\substack{p \in PD_{rr} \cap P_{2^+} \\ k \in B_p}} \left(c_{j,p,k,t} - \sum_{(k,l) \in EP_p} w_{i,j,p,t,k,l} \right) \tag{11}$$

Internal values in a composite pattern must not be put into a register (e.g. the multiply value in a multiply-and-accumulate instruction):

$$\forall rr \in \mathcal{RS}, tp \in T, tr \in T, p \in P_{2^+},\ \forall (k,l) \in EP_p,\ \forall (i,j) \in E_1 \cup E_2,$$
$$r_{rr,i,tr} \leq 1 - w_{i,j,p,tp,k,l} \tag{12}$$

If they exist, the first operand (Equation 13) and the second operand (Equation 14) must be available when they are used[2]:

$$\forall (i,j) \in E_1,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{\substack{p \in PS1_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{13}$$

[2]Constraints 11–14 have been improved compared to [Eriksson et al. 2008; Eriksson and Kessler 2009].

$$\forall (i,j) \in E_2,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{\substack{p \in PS2_{rr} \cap P_1 \\ k \in B_p}} c_{j,p,k,t} \tag{14}$$

Transfers may only occur if the source value is available:

$$\forall i \in V,\ \forall t \in T,\ \forall rr \in \mathcal{RS}, \quad r_{rr,i,t} \geq \sum_{rq \in \mathcal{RS}} x_{i,rr,rq,t} \tag{15}$$

2.6.4 *Non-dataflow dependences.* Equation 16 ensures that non-dataflow dependences are not violated, adapted from [Gebotys and Elmasry 1993]:

$$\forall (i,j) \in E_m,\ \forall t \in T \quad \sum_{p \in P} \sum_{t_j=0}^{t} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i=t-L_p+1}^{t_{\max}} c_{i,p,1,t_i} \leq 1 \tag{16}$$

2.6.5 *Resources.* We must not exceed the number of available registers in a register bank at any time:

$$\forall t \in T,\ \forall rr \in \mathcal{RS}, \quad \sum_{i \in V} r_{rr,i,t} \leq R_{rr} \tag{17}$$

Condition 18 ensures that no resource is used more than once at each time slot:

$$\forall t \in T,\ \forall f \in \mathcal{F},$$

$$\sum_{\substack{p \in P_{2+} \\ o \in \mathbb{N}}} U_{p,f,o} s_{p,t-o} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} U_{p,f,o} c_{i,p,k,t-o} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} UX_{rr,rq,f} x_{i,rr,rq,t} \leq 1 \tag{18}$$

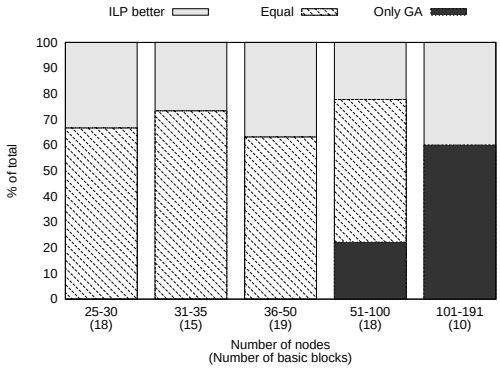
And, lastly, Condition 19 guarantees that we never exceed the issue width:

$$\forall t \in T, \quad \sum_{p \in P_{2+}} s_{p,t} + \sum_{\substack{p \in P_1 \\ i \in V \\ k \in B_p}} c_{i,p,k,t} + \sum_{\substack{i \in V \\ (rr,rq) \in (\mathcal{RS} \times \mathcal{RS})}} x_{i,rr,rq,t} \leq \omega \tag{19}$$

2.7 Experimental evaluation

We have compared our integer linear programming method to a genetic algorithm based heuristic ([Eriksson et al. 2008]). As input we used 80 basic blocks from the Mediabench benchmark suite [Lee et al. 1997]. The basic blocks were selected by taking all blocks with 25 or more IR nodes from the mpeg2 and jpeg encoding and decoding programs. The size of the largest basic block is 191 IR nodes. The target architecture is the two-clustered TI-C62x with small modifications.

The time limit for the algorithms was approximately 900 seconds per basic block. The solver is CPLEX 10.2 and the host machine is an Athlon X2 6000+ with 4 GB RAM. A summary of the results is shown in Figure 4; all DAGs solved by the integer linear programming method are optimal. The largest basic block that is solved by the integer linear programming method contains 142 IR nodes. After presolve 35302 variables and 21808 constraints remains, and the solution time is 672 seconds. We also saw basic blocks that are smaller in size (e.g. 76 IR nodes) and are not solved to optimality. Hence the time to optimally solve an instance does not only depend on the size of the DAG, but also on other characteristics of the problem, such as the amount of instruction level parallelism that is possible. For details on the genetic algorithm setup and results see [Eriksson et al. 2008].



4: Stacked bar chart showing a summary of the comparison between the integer linear programming and the genetic algorithm for basic blocks: *ILP better* means that ILP produces a schedule that is shorter than the one that GA produces, *Equal* means the schedules by ILP and GA have the same length, and *Only GA* means that GA finds a solution but ILP fails to do so. The integer linear programming method always produces an optimal result if it terminates.

3. INTEGRATED MODULO SCHEDULING

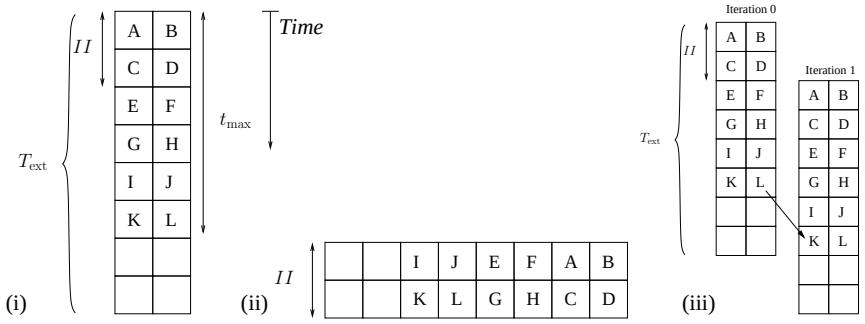
In this section we extend the integer linear programming model in Section 2 to modulo scheduling. We also show theoretical results on an upper bound for the number of schedule slots, and the results of an extensive evaluation.

3.1 Extending the model to modulo scheduling

Software pipelining [Charlesworth 1981] is an optimization for loops where the iterations of the loop are pipelined, i.e. consequtive iterations begin executing before the current one has finished. One well known kind of software pipelining is *modulo scheduling* [Rau and Glaeser 1981] where new iterations of the loop are issued at a fixed rate determined by the *initiation interval* (*II*). For every loop the initiation interval has a lower bound $MinII = \max(ResMII, RecMII)$, where *ResMII* is the bound determined by the available resources of the processor, and *RecMII* is the bound determined by the critical dependence cycle in the dependence graph describing the loop body. Methods for calculating *RecMII* and *ResMII* are well documented in e.g. [Lam 1988].

We note that a kernel can be formed from the schedule of a basic block by scheduling each operation modulo the initiation interval, see (i) and (ii) in Figure 5. The modulo schedules that we create have a corresponding iteration schedule, and by the *length* of a modulo schedule we mean the number of schedule slots ($t_{\max}$) of the iteration schedule. We also note that, since an iteration schedule is a potential basic block schedule, creating a valid modulo schedule only adds constraints compared to the basic block case.

First we need to model loop carried dependences by adding a distance to edges: $E_1, E_2, E_m \subset V \times V \times \mathbb{N}$. The element $(i, j, d) \in E$ represents a dependence from i to j which spans over d loop iterations. Obviously the graph is no longer a DAG since it may contain cycles. The only thing we need to do to include loop distances in the model is to change $r_{rr,i,t}$ to: $r_{rr,i,t+d \cdot II}$ in Equations 11, 13 and 14, and



5: An example showing how an acyclic schedule (i) can be rearranged into a modulo schedule (ii), A-L are target instructions in this example. (iii) An example showing why T_{ext} has enough time slots to model the extended live ranges. Here $d_{\max} = 1$ and $II = 2$ so any live value from Iteration 0 can not live after time slot $t_{\max} + II \cdot d_{\max}$ in the iteraton schedule.

modify Equation 16 to:

$$\forall (i,j,d) \in E_m,\ \forall t \in T_{\text{ext}} \quad \sum_{p \in P} \sum_{t_j=0}^{t-II \cdot d} c_{j,p,1,t_j} + \sum_{p \in P} \sum_{t_i=t-L_p+1}^{t_{\max}+II \cdot d_{\max}} c_{i,p,1,t_i} \leq 1 \qquad (20)$$

The initiation interval II must be a parameter to the integer linear programming solver. To find the best (smallest) initiation interval we must run the solver several times with different values of the parameter. A problem with this approach is that it is difficult to know when an optimal II is reached if the optimal II is not *RecMII* or *ResMII*; we will get back to this problem in Section 3.2.

The slots on which instructions may be scheduled are defined by $t_{\max}$, and we do not need to change this for the modulo scheduling extension to work. But when we model dependences spanning over loop iterations we need to add extra time slots to model that variables may be alive after the last instruction of an iteration is scheduled. This extended set of time slots is modeled by the set $T_{\text{ext}} = \{0, \ldots, t_{\max} + II \cdot d_{\max}\}$ where $d_{\max}$ is the largest distance in any of E_1 and E_2. We extend the variables in $x_{i,r,s,t}$ and $r_{rr,i,t}$ so that they have $t \in T_{\text{ext}}$ instead of $t \in T$, this is enough since a value created by an instruction scheduled at any $t \leq t_{\max}$ will be read, at latest, by an instruction $d_{\max}$ iterations later, see Figure 5(iii) for an illustration.

3.1.1 *Resource constraints.* The constraints in the previous section now only need a few further modifications to also do modulo scheduling. The resource constraints of the kind $\forall t \in T,\ \text{expr} \leq \text{bound}$ (Constraints 17–19) is modified to:

$$\forall t_o \in \{0, 1, \ldots, II-1\}, \quad \sum_{\substack{t \in T_{\text{ext}}: \\ t \equiv t_o (\text{mod } II)}} \text{expr} \leq \text{bound}$$

For instance, Constraint 17 becomes:

$$\forall t_o \in \{0, 1, \ldots, II-1\},\ \forall rr \in \mathcal{RS}, \quad \sum_{i \in V} \sum_{\substack{t \in T_{\text{ext}}: \\ t \equiv t_o (\text{mod } II)}} r_{rr,i,t} \leq R_{rr} \qquad (21)$$

Input: A graph of IR nodes $G = (V, E)$, the lowest possible initiation interval $MinII$, and the architecture parameters.
Output: Modulo schedule.
$MaxII = t_{\text{upper}} = \infty$;
$t_{\max} = MinII$;
while $t_{\max} \leq t_{\text{upper}}$ **do**
  Compute $soonest'$ and $latest'$ with the current $t_{\max}$;
  $II = MinII$;
  while $II < \min(t_{\max}, MaxII)$ **do**
    solve integer linear program instance;
    if solution found **then**
      if $II == MinII$ **then**
        return solution; *//This solution is optimal*
      fi
      $MaxII = II - 1$; *//Only search for better solutions.*
    fi
    $II = II + 1$
  od
  $t_{\max} = t_{\max} + 1$
od

6: Pseudocode for the integrated modulo scheduling algorithm.

Inequality 21 guarantees that the number of live values in each register bank does not exceed the number of available registers. If there are overlapping live ranges, i.e. when a value i is saved at t_d and used at $t_u > t_d + II \cdot k_i$ for some integer $k_i > 1$ the values in consecutive iterations can not use the same register for this value. We may solve this e.g. by doing variable modulo expansion [Lam 1988].

3.1.2 *Removing more variables.* As we saw in Section 2.5 it is possible to improve the solution time for the integer linear programming model by removing variables whose values can be inferred. Now we can take loop-carried dependences into account and find improved bounds:

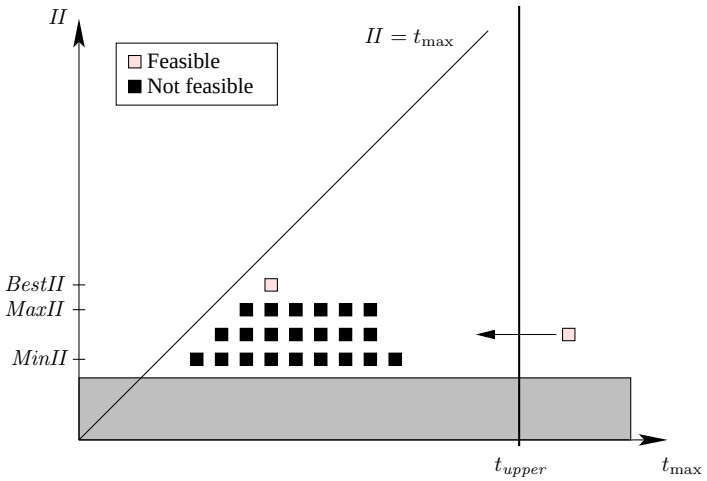
$$soonest(i) = \max \left\{ \begin{array}{l} soonest'(i), \\ \max_{\substack{(j,i,d)\in E \\ d\neq 0}} \left(soonest'(j) + L_{\min}(j) - II \cdot d\right) \} \end{array} \right\} \tag{22}$$

$$latest(i) = \max \left\{ \begin{array}{l} latest'(i), \\ \max_{\substack{(i,j,d)\in E \\ d\neq 0}} \left(latest'(j) - L_{\min}(i) + II \cdot d\right) \} \end{array} \right\} \tag{23}$$

With these new derived parameters we create

$$T_i = \{soonest(i), \ldots, latest(i)\} \tag{24}$$

that we can use instead of the set T for the t-index of variable $c_{i,p,k,t}$. Equations 22 and 23 differ from Equations 1 and 2 in two ways: they are not recursive and they need information about the initiation interval. Hence, $soonest'$ and $latest'$ can be calculated when $t_{\max}$ is known, before the integer linear program is run, and $soonest$ and $latest$ can be calculated parameters at solution time.



7: This figure shows the solution space of the algorithm. *BestII* is the best initiation interval found so far. For some architectures we can derive a bound, t_{upper}, on the number of schedule slots, $t_{\max}$, such that any solution to the right of t_{upper} can be moved to the left by a simple transformation.

3.2 The algorithm

Figure 6 shows the algorithm for finding a modulo schedule; this algorithm explores a two-dimensional solution space as depicted in Figure 7. The dimensions in this solution space are number of schedule slots ($t_{\max}$) and kernel size (II). Note that if there is no solution with initiation interval *MinII* this algorithm never terminates (we do not consider cases where $II > t_{\max}$). Later we will show how to make the algorithm terminate with the optimal result also in this case.

A valid alternative to this algorithm would be to set $t_{\max}$ to a fixed sufficiently large value and then solve for the minimal II. A problem with this approach is that the solution time of the integer linear program increases superlinearly with $t_{\max}$. Therefore we find that beginning with a low value of $t_{\max}$ and increasing it iteratively works best.

Our goal is to find solutions that are optimal in terms of throughput, i.e. to find the minimal initiation interval. An alternative goal is to also minimize code size, i.e. $t_{\max}$, since large $t_{\max}$ leads to long prologs and epilogs to the modulo scheduled loop. In other words: the solutions found by our algorithm can be seen as *pareto optimal* solutions with regards to throughput and code size where solutions with smaller code size but larger initiation intervals are found first.

3.2.1 *Theoretical properties.* In this section we will have a look at the theoretical properties of the algorithm in Figure 6 and show how the algorithm can be modified so that it finds optimal modulo schedules in finite time for a certain class of architectures.

Definition 3.1. We say that a schedule s is *dawdling* if there is a time slot $t \in T$ such that (a) no instruction in s is issued at time t, and (b) no instruction in s is running at time t, i.e. has been issued earlier than t, occupies some resource at

time t, and delivers its result at the end of t or later [Kessler et al. 2007].

Definition 3.2. The *slack window* of an instruction i in a schedule s is a sequence of time slots on which i may be scheduled without interfering with another instruction in s. And we say that a schedule is *n-dawdling* if each instruction has a slack window of at most n positions.

Definition 3.3. We say that an architecture is *transfer free* if all instructions except NOP must cover a node in the IR graph. I.e., no extra instructions such as transfers between clusters may be issued unless they cover IR nodes. We also require that the register file sizes of the architecture are unbounded.

LEMMA 3.4. *For a transfer free architecture every non-dawdling schedule for the data flow graph (V, E) has length*

$$t_{\max} \leq \sum_{i \in V} \hat{L}(i)$$

where $\hat{L}(i)$ is the maximal latency of any instruction covering IR node i (composite patterns need to replicate $\hat{L}(i)$ over all covered nodes).

PROOF. Since the architecture is transfer free only instructions covering IR nodes exist in the schedule, and each of these instructions is active at most $\hat{L}(i)$ time units. Furthermore we never need to insert dawdling NOPs to satisfy dependences of the kind $(i, j, d) \in E$; consider the two cases:

(a) $t_i \leq t_j$: Let $L(i)$ be the latency of the instruction covering i. If there is a time slot t between the point where i is finished and j begins which is not used for another instruction then t is a dawdling time slot and may be removed without violating the lower bound of j: $t_j \geq t_i + L(i) - d \cdot II$, since $d \cdot II \geq 0$.

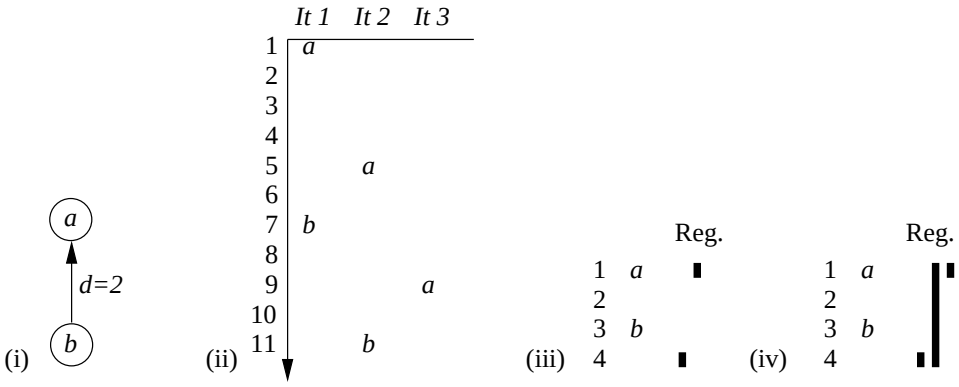
(b) $t_i > t_j$: Let $L(i)$ be the latency of the instruction covering i. If there is a time slot t between the point where j ends and the point where i begins which is not used for another instruction this may be removed without violating the upper bound of i: $t_i \leq t_j + d \cdot II - L(i)$. ($t_i$ is decreased when removing the dawdling time slot.) This is where we need the assumption of unlimited register files, since decreasing t_i increases the live range of i, possibly increasing the register need of the modulo schedule (see Figure 8 for such a case). □

COROLLARY 3.5. *An n-dawdling schedule for the data flow graph (V, E) has length*

$$t_{\max} \leq \sum_{i \in V} (\hat{L}(i) + n - 1) \quad .$$

Figure 8 shows an example that consists of a graph with two instructions, a and b, both with latency 1. The value produced by b is consumed by a two iterations later. Then, if the initiation interval is 4 the schedule shown in Figure 8 can not be shortened by 4 cycles, since this would increase the live range of b and hence increase the register pressure of the resulting modulo schedule.

LEMMA 3.6. *If a modulo schedule s with initiation interval II has an instruction i with a slack window of size at least $2II$ time units, then s can be shortened by II time units and still be a modulo schedule with initiation interval II.*



8: The graph in (i) can be modulo scheduled with initiation interval 4 as shown in (ii). If the schedule of an iteration is shortened by 4 cycles the register pressure of the corresponding modulo schedule kernel increases, see (iii) to (iv).

PROOF. If i is scheduled in the first half of its slack window the last II time slots in the window may be removed and all instructions will keep their position in the modulo reservation table. Likewise, if i is scheduled in the last half of the slack window the first II time slots may be removed. □

THEOREM 3.7. *For a transfer free architecture, if there does not exist a modulo schedule with initiation interval $\tilde{II}$ and $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$ there exists no modulo schedule with initiation interval $\tilde{II}$.*

PROOF. Assume that there exists a modulo schedule s with initiation interval $\tilde{II}$ and $t_{\max} > \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. Also assume that there exists no modulo schedule with the same initiation interval and $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. Then, by Lemma 3.4, there exists an instruction i in s with a slack window larger than $2\tilde{II} - 1$ and hence, by Lemma 3.6, s may be shortened by $\tilde{II}$ time units and still be a modulo schedule with the same initiation interval. If the shortened schedule still has $t_{\max} > \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$ it may be shortened again, and again, until the resulting schedule has $t_{\max} \leq \sum_{i \in V}(\hat{L}(i) + 2\tilde{II} - 1)$. □

COROLLARY 3.8. *We can guarantee optimality in the algorithm in Section 3.2 for transfer free architectures if, every time we find an improved II, we set $t_{\text{upper}} = \sum_{i \in V}(\hat{L}(i) + 2(II - 1) - 1)$.*

Until now we have assumed that the register file sizes are unbounded. Now we show how to allow bounded register file sizes by adding another assumption. The new assumption is that all loop carried dependences have distance no larger than 1.

LEMMA 3.9. *If there is a true data dependence $(b, a, d) \in E$ and a precedes b in the iteration schedule then the number of dawdling time slots between a and b is bounded by*

$$\omega_{a,b} \leq II \cdot d - L_b$$

where L_b is the latency of the instruction covering b.

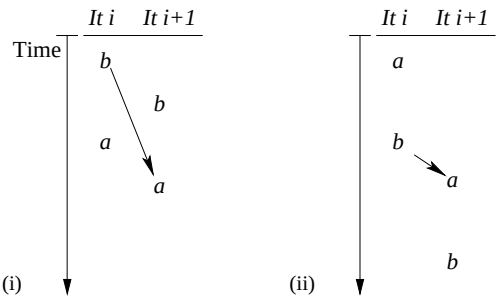


Fig. 9. Two cases: (i) b precedes a and (ii) a precedes b in the iteration schedule.

PROOF. The precedence constraint dictates

$$t_b + L_b \leq t_a + II \cdot d \tag{25}$$

If there are $\omega_{a,b}$ dawdling time slots between a and b in the iteration schedule then

$$t_b \geq t_a + \omega_{a,b} \tag{26}$$

Hence

$$t_a + \omega_{a,b} \leq t_b \leq t_a + II \cdot d - L_b \Rightarrow \omega_{a,b} \leq II \cdot d - L_b \quad \square$$

COROLLARY 3.10. *If $d_{\max} \leq 1$ then any transformation that removes a block of II dawdling time slots from the iteration schedule will not increase the register pressure of the corresponding modulo schedule with initiation interval II.*

PROOF. Consider every live range $b \rightarrow a$ that needs a register. First we note that the live range is only affected by the transformation if the removed block is between a and b.

If b precedes a in the iteration schedule (see Figure 9(i)) then removing a block of II nodes between b and a can only reduce register pressure.

If a precedes b in the iteration schedule (see Figure 9(ii)) then, by Lemma 3.9, assuming $L_b \geq 1$, there does not exist a removable block of size II between a and b in the iteration schedule. $\square$

With these observations we can change the assumption of unbounded register file sizes in Definition 3.3. The new assumption is that all loop carried dependences have distances smaller than or equal to 1. Furthermore, we can limit the increase in register pressure caused by removing a dawdling II-block:

COROLLARY 3.11. *Given an iteration schedule for a data flow graph $G = (V, E)$ the largest possible increase in register pressure of the modulo schedule with initiation interval II caused by removing dawdling blocks of size II is bounded by*

$$R_{increase} \leq \sum_{\substack{(b,a,d) \in E \\ d>1}} (d-1)$$

PROOF. Consider a live range $b \rightarrow a$ with loop carried distance $d > 1$. By Lemma 3.9 there are at most

$$\left\lfloor \frac{II \cdot d - L_b}{II} \right\rfloor < d - 1$$

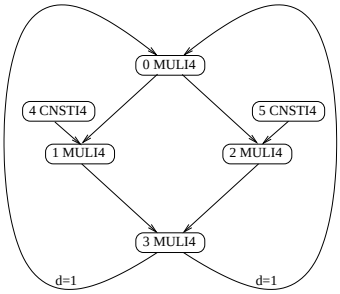


Fig. 10. A loop body with 4 multiplications. The edges between Node 3 and Node 0 are loop carried dependences with distance 1.

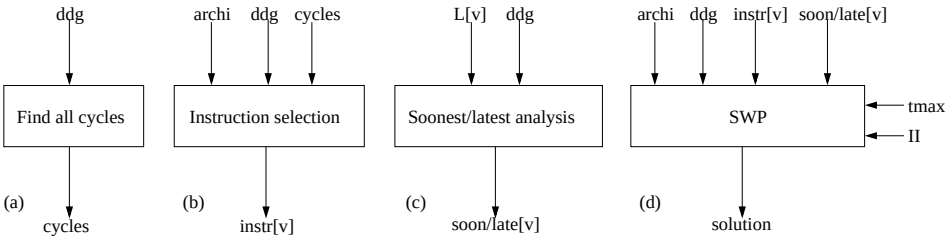
blocks of size *II* between a and b in the iteration schedule if a precedes b (if b precedes a there can be no increase in register pressure with the same reasoning as above). □

3.2.2 *A contrived example.* Let us consider an example that demonstrates how Corollary 3.8 can be used. Figure 10 shows a graph of an example program with four multiplications. Consider the case where we have a non-clustered architecture with one functional unit which can perform pipelined multiplications with latency 2. Clearly, for this example we have $RecMII = 6$ and $ResMII = 4$, but an initiation interval of 6 is impossible since IR-nodes 1 and 2 can not be issued at the same clock cycle. When we run the algorithm we quickly find a modulo schedule with initiation interval 7, but since this is larger than *MinII* the algorithm can not determine that it is an optimal solution. Now we can use Corollary 3.8 to find that an upper bound of 18 can be set on $t_{\max}$. If no improved modulo schedule is found where $t_{\max} = 18$ then the modulo schedule with initiation interval 7 is optimal. This example is solved to optimality in a few seconds by our algorithm.

3.3 Separating versus integrating instruction selection

In this section we will investigate and quantify the difference in code quality between our integrated code generation method and a method where instruction selection and cluster assignment are not integrated with the other phases. One way to do this is to first use a separate algorithm for cluster assignment and instruction selection. In this way we could define values for some of the solution variables in the integrated integer linear programming model and then solve the restricted instance. This would allow for a comparison of the achieved initiation interval for the less integrated method to the more integrated one. In this section we describe the details of how we did this and show the results of an extensive evaluation.

Figure 11 shows the components of the software pipelining method that we have developed. The integrated software pipelining model (d) takes a set of instructions, which are to be picked from, for each IR-node. The default set is the one which contains all instructions that can cover the operation that the IR-node represent. So making instruction selection separated is just a matter of supplying a set with only one instruction for each IR-node. Soonest-latest analysis (c) was described earlier, here it is extended so that the exact latency of each node of the graph can be given as input. This is useful for the cases where instruction selection has already been done. If the latencies are not known before the software pipelining phase we must use conservative values for these latencies. The integer linear programming



11: The components of the separated method.

model for the instruction selection phase (b) is basically the same model as the fully integrated one, but with the scheduling and register constraints removed and with a modified objective function that solves for minimal *MinII*. The cycle finding part (a) is used to aid in calculating *RecMII*.

3.3.1 *The instruction selection phase.* For instruction selection we use a smaller version of the integer linear programming model for the modulo scheduling phase. The constraints regarding scheduling and register allocation were stripped out and Constraints 6–9 are kept. To ensure that the instruction selection will work we add a constraint saying that for all edges $(a, b, d) \in E_1 \cup E_2$, if a writes to rr and b reads from $rs \neq rr$ there must be a transfer for this edge. Then we can bound *ResMII* by the following constraint (here with a simplified presentation):

$$\forall f \in \mathcal{F}, \quad \mathit{ResMII} \geq \sum_{i \in V} \text{instr. covering } i \text{ uses } f + \sum_{y \in \text{transfers}} y \text{ uses } f \tag{27}$$

When we calculate the lower bound on the initiation interval we need to take two things into account: the available resources and the cycles of the DDG. We implemented Tarjan's algorithm for listing all cycles in a graph [Tarjan 1973]. We ran this algorithm once on all data dependence graphs to generate all cycles; the time limit of the algorithm was set to 10 seconds[3], and if the limit was exceeded we stored all cycles found so far. For the cases where the enumeration algorithm exceeded the time limit, we had already found several thousand cycles. Continuing beyond this point may lead to finding more critical cycles, which could tighten the lower bound *RecMII*.

A cycle is represented by a set of all IR-nodes that are in it. Every cycle has a distance sum, that is the sum of distances of the edges that make up the cycle. Let $\mathrm{Cyc}(V, E)$ be the set of cycles defined by the graph with nodes V and edges E. And let $\mathrm{dist}(C)$ be the distance sum of cycle $C \in \mathrm{Cyc(V, E)}$. Let $\mathit{intern}[p] = \{a : \exists b, (a, b) \in E_p\}$ denote the set of inner nodes in pattern p, then *RecMII* can be bounded by

$$\forall C \in \mathrm{Cyc}(V, E), \quad \sum_{\substack{n \in C \\ p \in P \\ k \in (B_p - \mathit{intern}[p]) \\ t \in T}} L_p c_{n,p,k,t} \leq \mathit{RecMII} \cdot \mathrm{dist}(C) \tag{28}$$

[3]These 10 seconds are not included in the reported solution times in the evaluation section.

In the instruction selection phase the most important objective is to minimize the lower bound on the initiation interval. However, just minimizing the lower bound is not enough in many cases: when *MinII* is defined by *RecMII* and *ResMII* is significantly lower than *RecMII* the instruction selection solution may be very unbalanced with regards to resource allocation of the functional units. To address this problem we make the instruction selection phase minimize *MinII* as a first objective, and with this minimal *MinII* also minimize *ResMII*. Assuming that $MinII < M$ (we used $M = 100$ in our experiments) this can be expressed as:

$$\min \; M \cdot MinII + ResMII \tag{29}$$

subject to:

$$MinII \geq ResMII \, MinII \geq RecMII \tag{30}$$

The advantage of adding *ResMII* to the objective function of the minimization problem is that resources will be used in a more balanced way for the cases where the *MinII* is dominated by *RecMII*. This better balance makes the scheduling step easier, but the more complicated objective function in the instruction selection phase leads to a much increased memory usage. Even though the increased memory usage of the instruction selection phase makes the machine run out of memory for some instances the results are on average better than when the simple objective function is used.

3.3.2 *Results.* The experiments use data dependence graphs generated by the st200cc compiler with optimization level -O3. Input to the compiler are programs from the Spec2000, Spec2006, Mediabench and Ffmpeg benchmark [Touati 2009]. For solving the integer linear programming instances we use CPLEX 10.2 running on an Intel Core I7 950. The optimization goal is to minimize the initiation interval. After removing all graphs which include instructions that are too specialized for our model (e.g. shift-add instructions) and all duplicate graphs we have 1151 graphs left that have 60 or fewer IR-nodes. We have used two target architectures: a single cluster and a double cluster variant of TI-C62x. The double clustered variant has multiply-and-add instructions and includes transfers and spill instructions. The single cluster variant does not have composites, transfers or spill instructions.

Figure 12 summarizes the comparison between the integrated and the separated method, both with a time limit of 2 minutes. For the single-cluster architecture (a) we see that the cases where the integrated method is better than the separated one are rare; it only happens for 6% of the instances. The reason for this is that it is relatively easy to find a good instruction selection for this simple architecture, hence not much is gained by integrating instruction selection. When we look at the results for the two-cluster architecture (b) we find that the integrated method beats the separated one in 27% of the instances. It is more common that the integrated method beats the separated method when the graphs are small. However, as the size of the graphs become larger the increased complexity makes the integrated method time out often without finding any solution at all, while the simpler separated method finds a solution. The few cases where both methods find a solution, and the separated solution is better, is explained by the fact that the separated method reaches larger $t_{\max}$ before it times out, so it explores a larger part of the solution space in terms of iteration schedule length. Table I summarizes the results for

Table I. Average values of *IntII*/*SepII* for the instances where Int. is better than Sep.

Architecture	1-10	11-20	21-30	31-40	41-50	51-60	all
Single cluster	.84	.88	.84	.95	.95	.98	.88
Double cluster	.69	.81	.84	.85	.85	.93	.79

the instances where the integrated method finds a solution that is better than the one found by the separated method. The values shown are average values of *IntII*/*SepII*, where *IntII* and *SepII* are the found initiation intervals of the integrated and the separated methods respectively. These values quantifies how much better the integrated method is compared to the separated method. For instance: if the size of the kernel is halved then the throughput of the loop is doubled assuming that the number of iterations is large enough (so that the prologue and epilogue of the loop are negligible). The average value of *IntII*/*SepII* for the two-cluster architecture is 0.79, i.e. the average asymptotic speedup for the loops is $1/0.79 = 1.26$.

Figure 13 shows the results of an experiment with a variation of the two-cluster architecture where the two cross paths of the TI-C62x is replaced by a bus, and the number of registers per cluster is reduced from 16 to 8, for this test we increased the time limit to 30 minutes and limited the number of instances to 50 per size range. Now, because of the increased time limit, both methods perform better, but there are still quite many instances for which both methods fail to find a solution, these must be solved by a cheaper heuristic.

4. RELATED WORK

In this section we list some of the related work in the area of code generation both for basic blocks and for loops.

4.1 Integrated code generation for basic blocks

4.1.1 *Optimal methods.* Kästner [2001] has developed a retargetable phase coupled code generator which can produce optimal schedules by solving a generated integer linear program in a postpass optimizer. Two integer linear programming models are given. The first one is time based, like ours, and assigns events to points in time. The second formulation is order based where the order of events is optimized, and the assignment to points in time is implicit. The advantage of the second model is that it can be flow-based such that the resources flow from one instruction to the next, and this allows for more efficient integer linear program models in some cases but is less suitable for integrating other tasks than scheduling and resource allocation.

Wilken et al. [2000] have presented an integer linear programming formulation for instruction scheduling for basic blocks. They also discuss how DAG transformations can be used to make the problem easier to solve without affecting the optimality of the solution. The machine model which they use is rather simple.

Wilson et al. [1994] created an integer linear programming model for the integrated code generation problem with included instruction selection. This formulation is limited to non-pipelined, single issue architectures.

A constraint programming approach to optimal instruction scheduling of su-

perblocks[4] for realistic architectures was given by Malik et al. [2008]. The method was shown to be useful also for very large superblocks after a preprocessing phase which prunes the search space in a safe way.

Another integer linear programming method by Chang et al. [1997] performs integrated scheduling, register allocation and spill code generation. Their model targets non-pipelined, multi-issue, non-clustered architectures. Spill code is integrated by preprocessing the DAG in order to insert nodes for spilling where appropriate.

Winkel has presented an optimal method based on integer linear programming formulation for global scheduling for the IA-64 architecture [2004] and shown that it can be used in a production compiler [2007]. Much attention is given to how the optimization constraints should be formulated to make up a tight solution space that can be solved efficiently.

The integer linear programming model presented here for integrated code generation is an extension of the model by Bednarski and Kessler [2006]. Several aspects of our model are improved compared to it: Our model works with clustered architectures which have multiple register banks and data paths between them. Our model handles transfer instructions, which copy a value from one register bank to another (transfers do not cover an IR node of the DAG). Another improvement is that we can handle general dependences. We also remodeled the data flow dependences to work with the r variable

Within the Optimist project an integrated approach to code generation for clustered VLIW processors has been investigated by Kessler and Bednarski [2006]. Their method is based on dynamic programming and includes safe pruning of the solution space by removing comparable partial solutions.

4.1.2 *Heuristic methods.* Hanono and Devadas present an integrated approach to code generation for clustered VLIW architectures in the AVIV framework [1998]. Their method builds an extended data flow graph representation of a basic block which explicitly represents all alternatives for implementation, and then uses a branch-and-bound heuristic for selecting one alternative.

Lorenz et al. [2004] implemented a genetic algorithm for integrated code generation for low execution time. This genetic algorithm includes instruction selection, scheduling and register allocation in a single optimization problem. It also takes the subsequent address code generation, with address generation units, into account. In a preprocessing step additional IR nodes are inserted in the DAG which represent possible explicit transfers between register files.

Other notable heuristic methods that integrate several phases of code generation for clustered VLIW have been proposed by Kailas et al. [2001], Özer et al. [1998], Leupers [2000] and Nagpal and Srikant [2004].

4.2 Integrated software pipelining

4.2.1 *Optimal methods.* An enumeration approach to software pipelining, based on dynamic programming, was given by Vegdahl [1992]. In this algorithm the dependence graph of the original loop body is replicated by a given factor, with extra dependences to the new nodes inserted accordingly. The algorithm then

[4]A superblock is a block of code that has multiple exit points but only one entry point.

creates a compacted loop body in which each node is represented once, thus the unroll factor determines how many iterations a node may be moved. This method does not include instruction selection and register allocation.

Blachot et al. [2006] have given an integer linear programming formulation for integrated modulo scheduling and register assignment. Their method, named Scan, is a heuristic which searches the solution space by solving integer linear programming instances for varying initiation intervals and numbers of schedule slots in a way that resembles our algorithm in Section 3.2. Their presentation also includes an experimental characterization of the search space, e.g. how the number of schedule slots and initiation intervals affects tractability and feasibility of the integer linear programming instance.

Yang et al. [2002] presented an integer linear programming formulation for rate- and energy-optimal modulo scheduling on an Itanium-like architecture, where there are fast and slow functional units. The idea is that instructions that are not critical can be assigned to the slow, less energy consuming, functional units thereby optimizing energy use. Hence, this formulation includes a simple kind of instruction selection.

Ning and Gao [1993] present a method for non clustered architectures where register allocation is done in two steps, the first step assigns temporary values to buffers and the second step does the actual register allocation. Our method is different in that it avoids the intermediate step. This is an advantage when we want to support clustered register banks and integrate spill code generation.

Altman et al. [1995] presented an optimal method for simultaneous modulo scheduling and mapping of instructions to functional units. Their method, which is based on integer linear programming, has been compared to a branch and bound heuristic by Ruttenberg et al. [1996].

Fimmel and Müller [2002] do optimal modulo scheduling for pipelined non-clustered architectures by optimizing a rational initiation interval. The initiation interval is a variable in the integer linear programming formulation, which means that only a single instance of the problem needs to be solved as opposed to the common method of solving with increasingly large initiation intervals.

Eichenberger et al. [1996] have formulated an integer linear programming model for minimizing the register pressure of a modulo schedule where the modulo reservation table is fixed.

Nagarakatte and Govindarajan [2007] formulated an optimal method for integrating register allocation and spill code generation. These formulations work only for non-clustered architectures and do not include instruction selection.

Eisenbeis and Sawaya [1996] describe an integer linear programming method for integrating modulo scheduling and register allocation. Their method gives optimal results when the number of schedule slots is fixed.

Fan et al. [2005] studied the problem of synthesizing loop accelerators with modulo scheduling. In their problem formulation the initiation interval is given and the optimization problem is to minimize the hardware cost. They present optimal methods based on integer linear programming and branch and bound algorithms. They also show and evaluate several methods for decomposing large problem instances in to separate subproblems to increase tractability at the cost of global

optimality.

4.2.2 *Heuristic methods.* Fernandes [1998] was the one who first described *clustered* VLIW architectures, and Fernandes et al. [1999] gave a heuristic method for modulo scheduling for such architectures. The method is called *Distributed modulo scheduling* (DMS) and is shown to be effective for up to 8 clusters. DMS integrates modulo scheduling and cluster partitioning in a single phase. The method first tries to put instructions that are connected by true data dependences on the same cluster. If that is not possible transfer instructions are inserted or the algorithm backtracks by ejecting instructions from the partially constructed schedule.

Huff [1993] was the first to create a heuristic modulo scheduling method that schedules instructions in a way that minimizes life times of intermediate values. The instructions are given priorities based on the number of slots on which they may be scheduled and still respect dependences. The algorithm continues to schedule instructions with highest priority either early or late, based on heuristic rules. If an instruction can not be scheduled, backtracking is used, ejecting instructions from the partial schedule.

Another notable heuristic, which is not specifically targeted for clustered architectures, is due to Llosa et al. [1995; 1996]. This heuristic, called Swing modulo scheduling, simultaneously tries to minimize the initiation interval and register pressure by scheduling instructions either early or late.

The heuristic by Altemose and Norris [2001] does register pressure responsive modulo scheduling by inserting instructions in such a way that known live ranges are minimized.

Stotzer and Leiss [1999] presented a backtracking heuristic for modulo scheduling after instruction selection for Texas Instruments C6x processors.

Nystrom and Eichenberger [1998] presented a heuristic method for cluster assignment as a prepass to modulo scheduling. Their machine model assumes that all transfer instructions are explicit. The clustering algorithm prioritizes operations in critical cycles of the graph, and tries to minimize the number of transfer instructions while still having high throughput. The result of the clustering prepass is a new graph where operations are assigned to clusters and transfer nodes are inserted. Their experimental evaluation shows that, for architectures with a reasonable number of buses and ports, the achieved initiation interval is most of the time equal to the one achieved with the corresponding fully connected, i.e. non-clustered, architecture, where more data ports are available.

A heuristic method for integrated modulo scheduling for clustered architectures was presented by Codina et al. [2001]. The method, which also integrates spill code generation is shown to be useful for architectures with 4 clusters.

Pister and Kästner [2005] presented a retargetable method for postpass modulo scheduling implemented in the Propan framework.

4.2.3 *Theoretical results.* Touati [2007] presented several theoretical results regarding the register need in modulo schedules. One of the results shows that, in the absence of resource conflicts, there exists a finite schedule duration ($t_{\max}$ in our terminology) that can be used to compute the minimal periodic register sufficiency of a loop for all its valid modulo schedules. Theorem 3.7 in this paper is related

to this result of Touati. We assume unbounded register files and identify an upper bound on schedule duration, in the presence of resource conflicts.

5. CONCLUSIONS

We have studied the problem of integrated code generation for clustered architectures. The phases that are integrated are: cluster assignment, instruction selection, scheduling, register allocation and spilling. An algorithm was presented for the basic block case, and then we showed how it is extended to modulo scheduling.

The work presented in this paper is different from the ones mentioned in Section 4 in that it aims to produce provably optimal modulo schedules, also when the optimal initiation interval is larger than $MinII$, and in that it also integrates cluster assignment and instruction selection in the formulation. Our algorithm for modulo scheduling iteratively considers schedules with increasing number of schedule slots. A problem with such an iterative method is that, if the initiation interval is not equal to the lower bound, there is no way to determine whether the found solution is optimal or not. We have proven that, for a class of architectures that we call transfer free, we can set an upper bound on the schedule length. I.e. we can prove when a found modulo schedule with an initiation interval larger than the lower bound is optimal.

Creating an integer linear programming formulation for clustered architectures is more difficult than for the non-clustered case since the common method of modeling live ranges simply as the time between definition and use cannot be applied. Our formulation handles live ranges by explicitly assigning values to register banks for each time slot. This increases the size of the solution space, but we believe that this extra complexity is unavoidable and inherent to the problem of integrating cluster assignment and instruction selection with the other phases.

We have also shown that optimal spilling is closely related to optimal register allocation when the register files are clustered. In fact, optimal spilling is as simple as adding an additional virtual register file representing memory and have transfer instructions to and from this register file corresponding to stores and loads.

In our experimental evaluation we have shown that for the basic block case we can optimally generate code for DAGs with up to 142 IR nodes in less than 900 seconds. But we also saw that in some cases with only 76 IR nodes the integer linear programming model was not successful. For modulo scheduling we compare the integrated method to one in which instruction selection and cluster assignment is done in a separate phase. Our experiments show that the integrated method rarely results in better results than the separated for the single cluster architecture, but for the double cluster architecture the integrated method beats the separated one in 27% of the cases and in these cases, assuming a large number of iterations, the average speedup is 26%. The results of these experiments are important because we try to find out if the integration *can* be beneficial: both steps in the separated method are locally optimal and the integrated method is globally optimal. Showing how often the integrated method is better than the separated one is interesting also from a theoretical perspective.

ACKNOWLEDGMENTS

We thank Sid-Ahmed-Ali Touati for reading and commenting on an earlier version of this paper, and for providing the data dependence graphs used in the modulo scheduling evaluation. Oskar Skoog made the first implementation of the genetic algorithm used for comparison in this paper. We thank all reviewers of this and previous papers for constructive comments.

REFERENCES

Altemose, G. and Norris, C. 2001. Register pressure responsive software pipelining. In *Proc. of the ACM Symp. on Applied Computing (SAC'01)*. ACM, New York, 626–631.

Altman, E. R., Govindarajan, R., and Gao, G. R. 1995. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, 139–150.

Bednarski, A. and Kessler, C. W. 2006. Optimal integrated VLIW code generation with integer linear programming. In *European Conf. on Parallel Computing (Euro-Par'06)*. Springer, Berlin, 461–472.

Blachot, F., de Dinechin, B. D., and Huard, G. 2006. SCAN: A heuristic for near-optimal software pipelining. In *Proc. of the European Conf. on Parallel Computing (Euro-Par'06)*. Springer, Berlin, 289–298.

Chang, C., Chen, C., and King, C. 1997. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers and Mathematics with Applications 34,* 9, 1–14.

Charlesworth, A. 1981. An approach to scientific array processing: The architectural design of the AP-120b/FPS-164 family. *Computer 14,* 9 (Sept.), 18–27.

Codina, J. M., Sánchez, J., and González, A. 2001. A unified modulo scheduling and register allocation technique for clustered processors. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'01)*. IEEE, Los Alamitos, 175–184.

Eichenberger, A. E., Davidson, E. S., and Abraham, S. G. 1996. Minimizing register requirements of a modulo schedule via optimum stage scheduling. *Int. J. Parallel Program. 24,* 2, 103–132.

Eisenbeis, C. and Sawaya, A. 1996. Optimal loop parallelization under register constraints. In *Proc. of the 6th Workshop on Compilers for Parallel Computers (CPC'96)*. 245–259.

Eriksson, M. 2009. Integrated software pipelining. Licentiate degree thesis, Linköping Studies in Science and Technology Thesis No. 1393, Linköping University, Sweden.

Eriksson, M. V. and Kessler, C. W. 2009. Integrated modulo scheduling for clustered VLIW architectures. In *Proc. of the Int. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'09)*. Springer, Berlin, 65–79.

Eriksson, M. V., Skoog, O., and Kessler, C. W. 2008. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *Proc. of the 11th Int. Workshop on Software & Compilers for Embedded Systems (SCOPES'08)*. ACM, New York, 11–20.

Fan, K., Kudlur, M., Park, H., and Mahlke, S. 2005. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-38)*. IEEE, Los Alamitos, 219–232.

Fernandes, M. M. 1998. A clustered VLIW architecture based on queue register files. Ph.D. thesis, University of Edinburgh.

Fernandes, M. M., Llosa, J., and Topham, N. 1999. Distributed modulo scheduling. In *Proc. of the 5th Int. Symp. on High Performance Computer Architecture (HPCA'99)*. IEEE, Los Alamitos, 130.

Fimmel, D. and Müller, J. 2002. Optimal software pipelining with rational initiation interval. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*. CSREA Press, 638–643.

Fisher, J. A. 1983. Very long instruction word architectures and the ELI-512. In *Proc. of the 10th Annual Int. Symp. on Computer Architecture (ISCA'83)*. ACM, New York, 140–150.

Gebotys, C. and Elmasry, M. 1993. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12,* 9 (Sept.), 1266–1278.

Hanono, S. and Devadas, S. 1998. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proc. of the 35th Annual Conf. on Design Automation (DAC'98)*. ACM, New York, 510–515.

Huff, R. A. 1993. Lifetime-sensitive modulo scheduling. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'93)*. ACM, New York, 258–267.

Kailas, K., Ebcioglu, K., and Agrawala, A. 2001. CARS: A new code generation framework for clustered ILP processors. In *Proc. of the 7th Int. Symp. on High-Performance Computer Architecture (HPCA'01)*. IEEE, Los Alamitos, 133–143.

Kästner, D. 2001. Propan: A retargetable system for postpass optimisations and analyses. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'00)*. Springer, Berlin, 63–80.

Kessler, C., Bednarski, A., and Eriksson, M. 2007. Classification and generation of schedules for VLIW processors. *Concurrency and Computation: Practice and Experience 19,* 18, 2369–2389.

Kessler, C. W. and Bednarski, A. 2006. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience 18,* 11, 1353–1390.

Lam, M. 1988. Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Not. 23,* 7, 318–328.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Int. Symp. on Microarchitecture (MICRO-30)*. IEEE, Los Alamitos, 330–335.

Leupers, R. 2000. Instruction scheduling for clustered VLIW DSPs. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'00)*. IEEE, Los Alamitos, 291.

Llosa, J., Gonzalez, A., Ayguade, E., and Valero, M. 1996. Swing modulo scheduling: A lifetime-sensitive approach. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques (PACT'96)*. IEEE, Los Alamitos, 80–86.

Llosa, J., Valero, M., Ayguadé, E., and González, A. 1995. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*. IEEE, Los Alamitos, 350–360.

Lorenz, M. and Marwedel, P. 2004. Phase coupled code generation for DSPs using a genetic algorithm. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE'04)*. IEEE, Los Alamitos, 1270–1275.

Malik, A. M., Chase, M., Russell, T., and van Beek, P. 2008. An application of constraint programming to superblock instruction scheduling. In *Proc. of the 14th Int. Conf. on Principles and Practice of Constraint Programming*. Springer, Berlin, 97–111.

Nagarakatte, S. G. and Govindarajan, R. 2007. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *Proc. of the 16th Int. Conf. on Compiler Construction*. Springer, Berlin, 126–140.

Nagpal, R. and Srikant, Y. N. 2004. Integrated temporal and spatial scheduling for extended operand clustered VLIW processors. In *Proc. of the 1st Conf. on Computing Frontiers*. ACM, New York, 457–470.

Ning, Q. and Gao, G. R. 1993. A novel framework of register allocation for software pipelining. In *Proc. of the 20th ACM Symp. on Principles of Programming Languages (POPL'93)*. ACM, New York, 29–42.

Nystrom, E. and Eichenberger, A. E. 1998. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st annual ACM/IEEE Int. Symp. on Microarchitecture (MICRO-31)*. IEEE, Los Alamitos, 103–114.

Ozer, E., Banerjia, S., and Conte, T. M. 1998. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *Proc. of the 31st Annual ACM/IEEE Int. Symp. on Microarchitecture (MICRO-31)*. IEEE, Los Alamitos, 308–315.

Pister, M. and Kästner, D. 2005. Generic software pipelining at the assembly level. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES'05)*. ACM, New York, 50–61.

Rau, B. R. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. of the 27th Annual Int. Symp. on Microarchitecture (MICRO-27)*. ACM, New York, 63–74.

Rau, B. R. and Glaeser, C. D. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl. 12,* 4, 183–198.

Ruttenberg, J., Gao, G. R., Stoutchinin, A., and Lichtenstein, W. 1996. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'96)*. ACM, New York, 1–11.

Stotzer, E. and Leiss, E. 1999. Modulo scheduling for the TMS320C6x VLIW DSP architecture. *SIGPLAN Not. 34,* 7, 28–34.

Tarjan, R. E. 1973. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput. 2,* 3, 211–216.

Texas Instruments Incorporated 2000. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments Incorporated.

Touati, S. 2009. Data dependence graphs from Spec, Mediabench and Ffmpeg benchmark suites. Personal communication.

Touati, S.-A.-A. 2007. On periodic register need in software pipelining. *IEEE Trans. Comput. 56,* 11, 1493–1504.

Vegdahl, S. R. 1992. A dynamic-programming technique for compacting loops. In *Proc. of the 25th annual Int. Symp. on Microarchitecture (MICRO-25)*. IEEE, Los Alamitos, 180–188.

Wilken, K., Liu, J., and Heffernan, M. 2000. Optimal instruction scheduling using integer programming. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, 121–133.

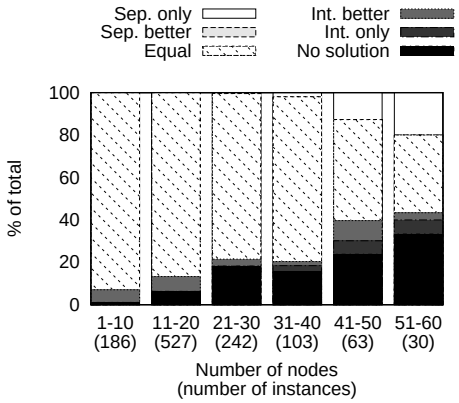
Wilson, T. C., Grewal, G. W., and Banerji, D. K. 1994. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *Proc. of the Int. Conf. on Computer Design (ICCD'94)*. IEEE, Los Alamitos, 581–586.

Winkel, S. 2004. Optimal global instruction scheduling for the Itanium processor architecture. Ph.D. thesis, Universität des Saarlandes.

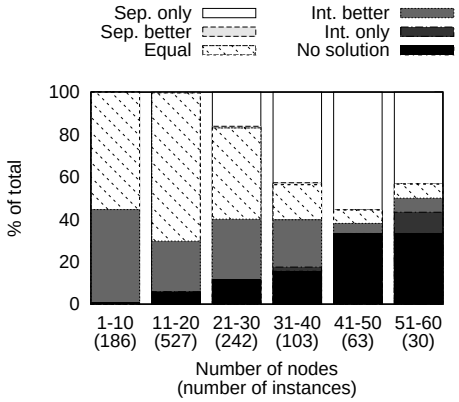
Winkel, S. 2007. Optimal versus heuristic global code scheduling. In *Proc. of the 40th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-40)*. IEEE, Los Alamitos, 43–55.

Yang, H., Govindarajan, R., Gao, G. R., and Theobald, K. B. 2002. Power-performance trade-offs for energy-efficient architectures: A quantitative study. In *Proc. of the IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'02)*. IEEE, Los Alamitos, 174.

Received June 2009; revised January 2010; accepted March 2010



(a) Single cluster.



(b) Double cluster.

12: Results of comparison between the separated and fully integrated version for the two architectures. The time limit is 2 minutes.

13: Results of comparison between the separated and fully integrated version for the clustered architecture. The time limit is 30 minutes and the number of instances is limited to 50 in this chart. For the cases where Int. finds a solution that is better than one found by Sep. the average value of $IntII/SepII$ is 0.82.