Energy-Optimal Integrated VLIW Code Generation

Andrzej Bednarski and Christoph Kessler
PELAB, Dept. of Computer Science, Univ. of Linköping, Sweden
{andbe,chrke}@ida.liu.se

Abstract

Optimal integrated code generation is a challenge in terms of problem complexity, but it provides important feedback for the resource-efficient design of embedded systems and is a valuable tool for the assessment of fast heuristics for code generation. We present a method for energy optimal integrated code generation for generic VLIW processor architectures that allows to explore trade-offs between energy consumption and execution time.

1 Introduction

Power dissipation in embedded systems is of serious concern especially for mobile devices that run on batteries. There are various approaches in embedded processor design that aim at reducing the energy consumed, and most of these have strong implications for power-aware code generation.

Voltage scaling reduces the power consumption by reducing voltage and clock frequency. The processor can be switched to such a power-saving mode in program regions where speed is of minor importance, such as waiting for user input. This technique is only applicable to coarse-grained regions of the program because transitions between modes have a non-negligible cost.

Clock gating denotes hardware support that allows to switch off parts of a processing unit that are not needed for a certain instruction. For instance, for integer operations, those parts of the processor that only deal with floatingpoint arithmetics can be switched off. Deactivation and reactivation require some (small) additional amount of energy, though. This feature for fine-grained power management requires optimizations by the code generator that avoid long sequences of repeated activations and deactivations by closing up instructions that largely use the same functional units. The method described in this paper will take this feature into account.

Pipeline gating reduces the degree of speculative execution and thus the utilization of the functional units. And there are several further hardware design techniques that exploit a trade-off between speed and power consumption. See [2] for an overview. Other factors that influence power dissipation are rather a pure software issue:

Memory accesses contribute considerably to power dissipation. Any power-aware code generator must therefore aim at reducing the number of memory accesses, for instance by careful scheduling and register allocation to minimize spill code, or by cyclic register allocation techniques for loops, such as register pipelining [16].

Switching activities on buses at the bit level are also significant. In CMOS circuits, power is dissipated when a gate output changes from 0 to 1 or vice versa. Hence, bit toggling on external and internal buses should be reduced. Hardware design techniques such as low-power bus encoding

may be useful if probability distributions of bit patterns on the buses are given [14]. However, the code generator can do much more. For instance, the bit patterns for subsequent instruction words (which consist of opcodes, register addresses and immediates) should not differ much, *i.e.* have a small Hamming distance. This implies constraints for instruction selection and for register assignment as well as for the placement of data in memory. Moreover, the bit patterns of the instruction addresses (*i.e.*, the program counter value) do matter. Even the contents of the registers accessed have an influence on power dissipation. Beyond the Hamming distance, the *weight* (*i.e.* the number of ones in a binary word) may have an influence on power dissipation – positive or negative.

Instruction decoding and execution may take more or less energy for different instructions. This is a challenge for instruction selection if there are multiple choices. For instance, an integer multiplication by 2 may be replaced by a left shift by one or by an integer addition. Often, multiplication has a higher base cost than a shift operation [12]. Different instructions may use different functional units, which in turn can influence the number of unit activations/deactivations. Hence, resource allocation is a critical issue. In the multiplication example, if the adder is already "warm" but the shifter and multiplier are "cold", *i.e.* not used in the preceding cycle, the power-aware choice would be to use the adder, under certain conditions even if another addition competes for the adder in the same time slot.

Execution time in terms of the number of clock cycles taken by the program is directly related to the energy consumed, which is just the integral of power dissipation over the execution time interval. However, there is no clear correlation between execution time and e.g. switching activities [17]. There are further trade-offs such as between the number of cycles needed and the number of unit activations/deactivations: Using a free but "cold" functional unit increases parallelism and hence may reduce execution time, but also increases power dissipation. Such effects make this issue more tricky than it appears at a first glance.

In order to take the right decisions at instruction selection, instruction scheduling (including resource allocation) and register allocation (including register assignment), the code generator needs a *power model* that provides quite detailed information about the power dissipation behavior of the architecture, which could be given as a function of the instructions, their encoding, their resource usage, their parameters, and (if statically available) their address and the data values accessed. On the other hand, such a power model should abstract from irrelevant details and thus allow for a fast calculation of the expected power dissipation for a given program trace. Ideally, a power model should be applicable to an entire class of architectures, to enhance retargetability of a power-aware code generator.

The information for a power model can be provided in two different ways: by simulation and by measurements. Simulation-based approaches take a detailed description of the hardware as input and simulate the architecture with a given program at the microarchitectural level, cycle by cycle to derive the energy consumption. Examples for such simulators are SimplePower [22] and Wattch [1]. In contrast, measurement-based approaches assume a small set of factors that influence power dissipation, such as the width of opcodes, Hamming distances of subsequent instruction words, activations of functional units, etc., which are weighted by originally unknown parameters and summed up to produce the energy prediction for a given program trace. In order to calibrate the model for a given hardware system, an ampèremeter is used to measure the current that is

actually drawn for a given test sequence of instructions [18]. The coefficients are determined by regression analysis that takes the measurements for different test sequences into account. Such a statistical model is acceptable if the predicted energy consumption for an arbitrary program differs from the actual consumption only by a few percent. Recently, two more detailed, measurement-based power models were independently developed by Lee et al. [11] and by Steinke et al. [15], for the same processor, the ARM7TDMI; they report on energy predictions that are at most 2.5% and 1.7% off the actual energy consumption, respectively. We will use an adapted version of their power model as a basis for the energy optimizations described in this paper.

Higher level compiler optimization techniques may address loop transformations, memory allocation, or data layout. For instance, the memory layout of arrays could be modified such that consecutive accesses traverse the element addresses in a gray code manner. If available, some frequently accessed variables could be stored in small on-chip memory areas [12]. In this work, we assume that such optimizations are already done and we focus on the final code generation step.

Not all factors in a power model are known at compile time. For instance, instruction addresses may change due to relocation of the code at link or load time, and the values residing in registers and memory locations are generally not statically known, even though static analysis could be applied to predict e.g. equality of values at least in some cases.

We present a method for energy-aware integrated local code generation that allows to explore trade-offs between energy consumption and execution time for a generic VLIW processor architecture. Our framework can be applied in two ways: It can determine power-optimal code (for a given power model) and it can optimize for execution time given a user-specified energy budget for (parts of) the program. An integrated approach to code generation is necessary because the subproblems of instruction selection, instruction scheduling, resource allocation and register allocation depend on each other and should be considered simultaneously. This combined optimization problem is a very hard one, even if only the basic block scope is considered. Fortunately, the application program is often fixed in embedded systems, and the final production run of the compiler can definitely afford large amounts of time and space for optimizations. Finally, an optimal solution is of significance for the design of energy-efficient processors because it allows to evaluate the full potential of an instruction set design. Furthermore knowing the optimal solution allows to evaluate the quality of fast heuristics, as we will show later.

2 Target processor model

Our processor model is a generic VLIW processor with f functional units $U_1, ..., U_f$, one or more register sets, and one or more memory modules. With some limitations this is applicable to superscalar processors as well.

Time model The *issue width* ω of the processor is the maximum number of instructions that may be issued in the same clock cycle. Usually, $1 \leq \omega \leq f$. The *unit occupation time* o_i of a functional unit U_i is the number of clock cycles that U_i is occupied with executing an instruction before a new instruction can be issued to U_i. The *latency* ℓ_i of a unit U_i is the number of clock cycles taken by an instruction on U_i before the result is available. We assume that $o_i \leq \ell_i$. Different target instructions can specify other values for occupation time (o_i') and latency (ℓ_i') for unit U_i.

We assume that for every instruction that modifies parameters of a functional unit U_i, $o'_i \leq o_i$, $\ell'_i \leq \ell_i$ and $o'_i \leq \ell'_i$.

Power model We adopt a simple power model [11, 12]. that largely follows the measurement-based power models described above, which we generalize in a straightforward way for VLIW architectures. Our model assumes that the contribution of every instruction y to the total energy consumption consists of the following components: (i) a base cost *bcost*(y) that is independent of the context of the instruction, (ii) an overhead cost *ohcost*(y, y') that accounts for inter-instruction effects with the instruction y' that precedes y in the same field of the instruction word, such as bit toggling in the opcode fields, and (iii) an activation/deactivation cost ac_i that is paid if functional unit U_i is activated or deactivated.

ADML The structure (U_i, instruction set) and parameters (ω, ℓ_i, o_i, ...) of our generic architecture model can be specified in our XML-based architecture description markup language ADML [6].

3 Energy-optimal integrated code generation

We focus on code generation for basic blocks and extended basic blocks [13] where the data dependences among the IR operations form a directed acyclic graph (DAG) $G = (V, E)$. Let n denote the number of IR nodes in the DAG.

IR-level scheduling An *IR-schedule*, or simply *schedule*, of the basic block (DAG) is a bijective mapping $S : V \rightarrow \{1, ..., n\}$ describing a linear sequence of the n IR operations in V that is compliant with the partial order defined by E, that is, $(u, v) \in E \Rightarrow S(u) < S(v)$. A *partial schedule* of G is a schedule of a subDAG $G' = (V', E \cap (V' \times V'))$ induced by a subset $V' \subseteq V$ where for each $v' \in V'$ holds that all predecessors of v' in G are also in V'. A partial schedule of G can be extended to a (complete) schedule of G if it is prefixed to a schedule of the remaining DAG induced by $V - V'$.

Instruction selection Naive instruction selection maps each IR operation v to one of a set $\Psi(v)$ of equivalent, single target instructions: A *matching target instruction* $y \in \Psi(v)$ for a given IR operation v is a target processor instruction that performs the operation specified by v. An *instruction selection* Y for a DAG $G = (V, E)$ maps each IR operation $v \in V$ to a target instruction $y \in \Psi(v)$.

Our framework also supports the case that a single target instruction y covers a set χ of multiple IR operations, which is quite common for a low-level IR. This corresponds to a generalized version of tree pattern matching. The converse case of a single IR operation corresponding to multiple target instructions requires either lowering the IR or scheduling on the target level only.

Target-level scheduling A target instruction y may actually require time slots on several functional units. A *target-schedule* is a mapping s of the time slots in $\{U_1, ..., U_f\} \times \mathbb{N}_0$ to instructions such that $s_{i,j}$ denotes the instruction starting execution on unit U_i at time slot j. Where no instruction is started on U_i at time slot j, $s_{i,j}$ is defined as $\perp$ (idle). If an instruction $s_{i,j}$ produces a value that is used by an instruction $s_{i',j'}$, it must hold $j' \geq j + \ell_i$. Also, it must hold $j' \geq j'' + o_i$ where $s_{i',j''}$ is the latest instruction issued to $U_{i'}$ before $s_{i',j'}$. Finally, it must hold $|\{s_{i'',j'} \neq \perp, 1 \leq i'' \leq f\}| \leq \omega$.

In addition to this per-unit view of the target schedule s we are also interested in the *instruction*

word view σ^s that shows how instructions appear in the ω slots per cycle in the program. An instruction $\sigma^s_{i,j}$ corresponds to the i-th slot of the j-th instruction word in the schedule σ^s. $\sigma^s_{i,j} =$ NOP if not filled.

The *reference time* $\rho(s)$ of a target-schedule s is the last clock cycle where an instruction (including explicit NOPs) is issued in s to some functional unit. The *execution time* $\tau(s)$ of a target-schedule s is the number of clock cycles required for executing s, that is, $\tau(s) = \max_{i,j}\{j+\ell_i : s_{i,j} \neq \perp\}$.

A target schedule s is *time-optimal* if it takes not more time that any other target schedule for the DAG.

Some instructions issued at time $t \leq \rho(s)$ may not yet terminate at time $\rho(s)$, which means that the time slots $\rho(s)+1, ..., \tau(s)$ in s must be padded by NOPs to guarantee a correct program. In order to account for the energy contribution of these trailing NOP instructions we transform the base cost of all instructions y to cover successive NOPs: *basecost*$(y) =$ *bcost*$(y) -$ *bcost*(NOP). In particular, the transformed *basecost*(NOP) is 0. Thus the base cost for target schedule s is:

$$E_{bc}(s) = \tau(s) \cdot \textit{bcost}(\texttt{NOP}) + \sum_{\sigma^s_{i,j} \neq \texttt{NOP}} \textit{basecost}(\sigma^s_{i,j}).$$

The overhead cost for a target schedule s can be calculated as follows:

$$E_{oh}(s) = \sum_{i=0}^{\omega} \sum_{j=1}^{\tau(s)} ohcost(\sigma^s_{i,j}, \sigma^s_{i,j-1})$$

The activation/deactivation cost of s is:

$$E_{act}(s) = \sum_{i=1}^{f} \sum_{j=1}^{\tau(s)} ac_i \cdot \delta'(s_{i,j}, s_{i,j-1})$$

where $\delta'(a,b) =$0 if $(a \neq \perp \wedge b \neq \perp) \vee (a = \perp \wedge b = \perp)$, and 1 otherwise.

Then the total energy cost for s is $E(s) = E_{bc}(s) + E_{oh}(s) + E_{act}(s)$.

Register allocation A *register allocation* for a given target-schedule s of a DAG is a mapping r from the scheduled instructions $s_{i,j}$ to physical registers such that the value computed by $s_{i,j}$ resides in a register $r(s_{i,j})$ from time slot j and is not overwritten before its last use.

The register need of an optimal register allocation for a given target-schedule s can be computed in linear time.

If width or Hamming distance of subsequent register contents or of the bit patterns in the register indices do matter, a more sophisticated approach would be necessary to compute a power-optimal register assignment for a given schedule, e.g. by using resource flow graphs [3] or a graph coloring technique. Power dissipation effects due to bit switching on the instruction bus are not modeled and currently considered as negligible.

Basic method A naive approach to finding an optimal target schedule consists in the exhaustive enumeration of all possible target schedules, each of which can be generated by a combination of topological sorting of the DAG nodes with DAG pattern matching for instruction selection. As in topological sorting, the algorithm maintains a set of DAG nodes with indegree zero (ready for

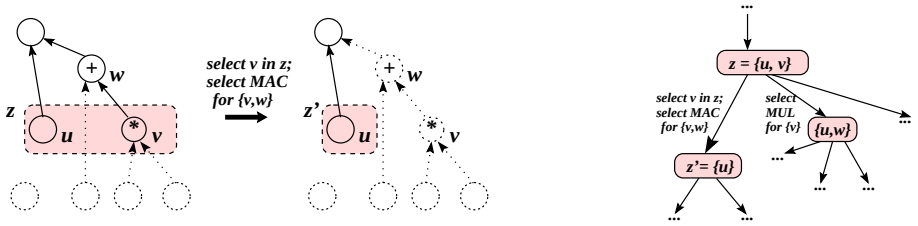

Figure 1: Basic method. Left hand side: situation when selecting $v \in z$ and a multiply-accumulate instruction that covers v. Right hand side: resulting selection tree.

selection), the *zero-indegree set*, which is initialized to the set z_0 of DAG leaves. The algorithm repeatedly does the following:

(1) A DAG node v from the current zero-indegree set z is selected. The algorithm selects a target instruction y that covers v (and possibly some further not yet scheduled DAG nodes). For irregular processors the algorithm checks the availability of operands for instruction y in the expected residence classes [6]. Let χ denote the set of DAG nodes covered by y.

(2) y is appended to the current target schedule and scheduled to the first possible slot on the unit(s) that y executes on.

(3) The nodes in χ are appended to the current IR schedule and removed from the DAG, which implies updating the indegrees of the parents of the nodes in χ. The zero-indegree set changes accordingly. If data placement is considered, the algorithm issues additional transfer operations in target schedule s [6].

(4) This process is continued until all DAG nodes have been scheduled.

An exhaustive enumeration of all possibilities for selecting the next node v from a zero-indegree list and of matching instructions y generates all possible IR-schedules and all possible *greedy* target-schedules of the DAG. The greedy compaction applied in step (2) is sufficient for pure time optimization because any schedule can be converted to a greedy schedule without increasing the execution time [4]. However, if register need or power dissipation do matter, a non-greedy schedule may be better, for instance to defer usage of a functional unit by a few cycles to cluster activation periods. In this case, we modify the algorithm to defer execution by issuing additional `NOP`s. This is done by adding in step (1) an implicit `NOP` node to each zeroindegree set z that, if selected, causes step (2) to flush the current long instruction word. In combination with exhaustive enumeration, this allows to generate all possible target schedules.

The exhaustive enumeration of (IR) schedules produced by topological sorting implicitly builds a tree-like representation of all schedules of the DAG, called the *selection tree* (see Fig. 1), which is leveled. Each node of the selection tree corresponds to a scheduling situation in the above algorithm, *i.e.* an instance of a zero-indegree set of DAG nodes during topological sorting. A directed edge connects a node z to a node z' of a selection tree if there is a step in the selection process of topological sorting that produces the zero-indegree set z' from z.

In previous work [5] we pointed out that multiple instances of the same zero-indegree set may occur in the selection tree. For all these instances, the same set *scheduled*(z) of nodes in the same subDAG G_z of G below z has been scheduled. This leads to the idea that we could perhaps op-

timize locally among all the partial schedules corresponding to equal zero-indegree set instances, merge all these nodes to a single selection node and keep just one optimal partial schedule to be used as a prefix in future scheduling steps. In [7] we have shown that this optimization is valid when computing space-optimal schedules for a single-issue processor. When applying this idea to all nodes of a selection tree, the selection tree becomes a *selection DAG*. In the same way as the selection tree, the selection DAG is leveled, where all zero-indegree sets z that occur after having scheduled $l = |\mathit{scheduled}(z)|$ DAG nodes appear at level l in the selection DAG, see Fig. 2. This grouping of partial schedules is applicable to schedules that are *comparable* with respect to the optimization goal. Comparability also depends on the target architecture. The resulting compression of the solution space decreases considerably the optimization time and makes it possible to generate space-optimal schedules for DAGs of reasonable size [7].

Time profiles For time-optimal schedules, comparability of partial schedules requires a more involved definition. In previous work [5] we introduced the concept of a *time profile* that records for a target schedule s which operations are currently (at time $\rho(s)$) being executed and have not yet completed on every functional unit, which may influence future scheduling decisions.

The time profile P of a given target-schedule s is determined from s as follows [5]: Let $t = \rho(s) \geq 0$ denote the reference time of s. The profile P is obtained by concatenating (in reverse order) the DAG nodes (or `NOP`s where units are idle) corresponding to the ℓ_i latest entries $s_{i,t-\ell_i+1}, ..., s_{i,t}$ in s for the units U_i, $i = 1, ..., f$. Entries $s_{i,j}$ with $j < 0$ are regarded as $\perp$s. The *time reference point* of P in s is t. Hence, a time profile contains all the information required to decide about the earliest time slot where the node selected next can be scheduled.

Power profiles A *power profile* $\Pi(s) = (s_{t-1,1}, ..., s_{t-1,\omega},\ a_1, ..., a_f)$ for a target schedule s at reference time $t = \rho(s)$ contains the instructions $s_{t-1,k}$ issued in each slot k of the next-to-last instruction word in s at time $t-1$, and the *activity status* $a_i \in \{0, 1\}$ in the last filled slot of unit U_i in s, that is, at time t if some instruction or a definite `NOP` was already scheduled to unit U_i at time t, and $t-1$ otherwise.

The power profile thus stores all the information that may be necessary to determine the impact of scheduling steps at time t on power dissipation: the activity status of all functional units says whether a unit must be activated or deactivated, and the information about the preceding instructions allows to calculate inter-instruction power effects for instructions to be selected and scheduled at time t. Given the power profile and a new instruction y to be appended to the current target schedule s, the new power profile can be calculated incrementally.

We can thus associate with every target schedule s for G_z the accumulated energy $E_z(s)$ that was consumed by executing s from time 1 to $\tau(s)$ according to our power model. The goal is of course to find a target schedule s for the entire basic block that minimizes $E_\emptyset(s)$. If we optimize for energy only (and thus ignore the time requirements), two target schedules s_1 and s_2 for the same subDAG G_z are comparable with respect to their energy consumption if they have the same power profile, as they could be used interchangeably as a prefix in future scheduling decisions. Hence, the following compression theorem allows us to apply our dynamic programming framework to energy optimization:

Theorem 3.1 *For any two target schedules for the same subDAG G_z, s_1 with reference time $t_1 = \rho(s_1)$ and s_2 with reference time $t_2 = \rho(s_2)$, where $\Pi(s_1) = \Pi(s_2)$, the s_i with higher*

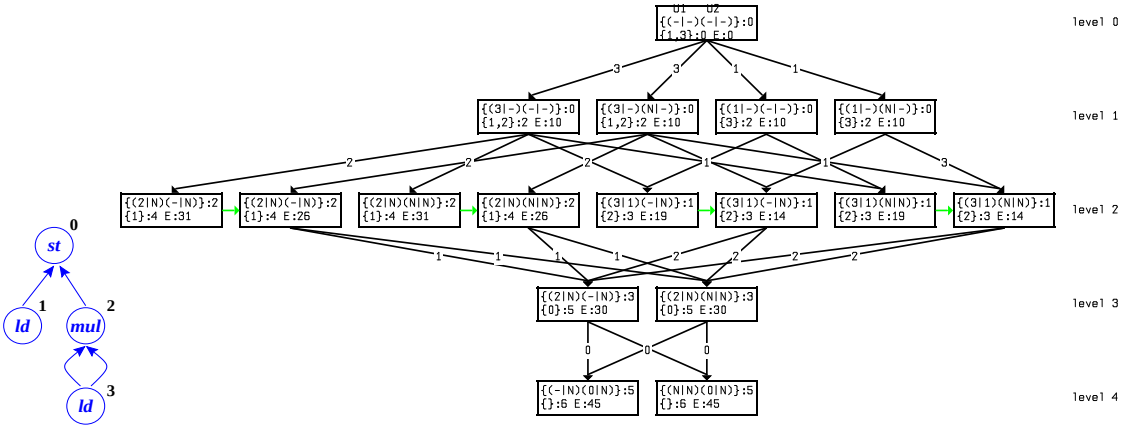

Figure 2: Solution space for squaring on a 2-issue architecture with 2 functional units $\ell = (2, 1)$, with base costs: 4 for st, 3 for others, 1 for NOP, and instruction overhead of 0.

accumulated energy consumption $E_z(s_i)$ can be thrown away, that is, needs not be considered in further scheduling steps, without losing optimality. If $E_z(s_1) = E_z(s_2)$, either s_1 or s_2 can be thrown away without losing optimality.

The theorem follows from the fact that all information that is not stored in the power profile, such as instructions issued at a time earlier than $t - 1$ or units active at a time earlier than $t - 1$, has no influence on the power dissipation in cycle t (reference time) according to our power model.

Construction of the solution space For energy-only optimization, an *extended selection node* (ESnode for short) $\eta = (z, \Pi)$ is characterized by a zeroindegree set z and a power profile $\Pi = \Pi(s)$ for some target schedule s of G_z that is locally energy-optimal among all target schedules for G_z with that power profile. The ESnode stores s as an attribute.

Fig. 2 represents the whole solution space for the matched DAG represented on the left side that computes `y=x*x`. The target architecture has two functional units U_1 and U_2 with latency $\ell = (2, 1)$ and activation cost $ac = (6, 5)$. The base cost for node `0` is 4, 3 for others, and 1 for `NOP`. Node `0` is to be executed on unit U_2 and other nodes on either U_1 or U_2. ESnodes are grouped according to their level (length of the IR schedule). The first row of each ESnode represents the power profile and its reference time $\rho(s)$. For example, for the leftmost ESnode at level 1, $\{(3|-)(-|-)\}$ means that node `3` (ld) matches instruction 3 and is scheduled on unit U_1 at the current time slot. The status of unit U_2 is empty (denoted by –). In other ESnodes, `N` denotes an explicit `NOP` inserted by the dynamic programming algorithm. Remark that we show explicitly the occupation status of each functional unit at reference time $t = \rho(s)$ and $t - 1$. Edges are annotated with nodes selected in the selection step of topological sorting (see Section 3). Edges that fall into a single ESnode indicate that the resulting partial solution nodes are equivalent. Unlabeled edges (horizontal) show an improvement of a partial solution, *i.e.* a partial ESnode is constructed that dissipates less energy than an equivalent ESnode that was already in the solution space; the resulting ESnode is the one with least energy dissipation. The second row in each ESnode represents the zero-indegree set (e.g., $1, 2$: nodes `1` and `2`), followed by the total execution time in terms of number of clock cycles (2 for the example node). Finally, we show the energy

required for the partial schedule (10).

We structure the solution space as a two-dimensional grid L, spanned by an energy axis and a length axis. In order to obtain a discrete solution space we partition the energy axis into intervals $[k \cdot \Delta E, (k+1) \cdot \Delta E[$ of suitable size ΔE and normalize the lower bounds of the intervals to the integers $k = 0, 1, 2, ...$ Grid entry $L(l, E)$ stores a list of all ESnodes that represent IR schedules of length l and accumulated energy consumption E. This structure supports efficient retrieval of all possible candidates for comparable partial solutions. When constructing the solution space, we proceed along the energy axis as driving axis, as energy is supposed to be the main optimization goal, while the length axis has secondary priority. The grid structure allows, by taking the precedence constraints for the construction of the partial solutions into account, to change the order of construction as far as possible such that the more promising solutions will be considered first while the less promising ones are set aside and reconsidered only if all the initially promising alternatives finally turn out to be suboptimal.

This is based on the property that the accumulated energy consumption never decreases if another instruction is added to an existing schedule. Most power models support this basic assumption; otherwise a transformation as in [6] can be applied to establish monotonicity for the algorithm.

Implementation and results The structuring and traversal order of the solution space allows us to optimize the memory consumption of the optimization algorithm. Once all partial solutions in an ESNode (E, l) have been expanded it can be removed, as it will never be looked up again. Our algorithm for finding an energy-optimal schedule appears to be practical up to size 30, see Table 1.

Heuristics for large problem instances Large DAGs require heuristic pruning of the solution space to cope with the combinatorial complexity. As a first attempt we control the number of variants generated from a scheduling situation, *i.e.* the number of ESnodes produced at each selection step. Instead of generating all possible selections we stop after N variants. Increasing the value of N results in better schedules with a slight computation time overhead. Using this heuristic decreases significantly computation times, that still present exponential behavior, and results in highly optimized code quality within 10% to optimal. We additionally implemented list scheduling (LS) and simulated annealing (SA) heuristics. For the results obtained with LS heuristic we observe an overhead of 173% on average and for SA 55%. This significant overhead for both heuristics is caused by that they do not consider using an already "warm" functional unit, nor the long delays for certain instructions resulting in switching on and off functional units often.

Table 1 shows the time requirements for finding an energy optimal schedule of our energy-only optimization algorithm on a collection of basic blocks taken from handwritten example programs and DSP benchmarks [6]. Measurements have been performed on a Linux PC with 1.6GHz AMD processor and 1.5GB RAM. Column BB refers to a basic block among different benchmark programs. The choice of a basic block was only its size, indicated in parenthesis. The second column reports the time in seconds for finding an energy-optimal schedule and its corresponding energy dissipation in energy unit (eU). If the algorithm run out of the time quantum (6 hours) it was interrupted, and indicated in the table by a dash. Columns 3-6 represent computation times for different values of N (1, 5, 10 and 25) and energy overheads compared to the optimal solution found in the optimal search. Finally, columns LS and SA indicate the energy dissipation and overhead obtained with a naive list scheduling (LS) and simulated annealing (SA) heuristics.

Table 1: Influence of heuristics on computation time and code quality.

BB	OPT		H1(s)		H5(s)		H10(s)		H25(s)		LS		SA	
	t(s)	(eU)	t(s)	o(%)	t(s)	o(%)	t(s)	o(%)	t(s)	o(%)	(eU)	o(%)	(eU)	o(%)
bb1 (22)	221.7	94	1.2	5	5.2	0	5.6	0	11.1	0	237	152	111	18
bb2 (22)	42.0	89	1.7	38	5.1	13	9.4	11	19.1	0	237	166	145	63
bb3 (22)	59.7	86	1.2	9	4.3	0	5.8	0	15.4	0	243	183	136	58
bb4 (23)	18.0	83	3.1	20	3.4	14	6.4	12	9.8	0	248	199	134	61
bb5 (25)	17.2	102	1.9	17	5.0	0	7.8	0	14.2	0	274	169	153	50
bb6 (25)	113.0	94	1.9	18	8.8	0	12.7	0	36.7	0	259	176	152	62
bb7 (25)	16.6	102	1.8	17	5.2	0	7.8	0	14.0	0	274	169	158	55
bb8 (27)	560.0	101	2.7	34	12.2	10	23.3	0	78.9	0	277	174	152	50
bb9 (30)	112.4	112	2.6	20	12.8	9	22.3	0	53.8	0	304	171	180	60
bb10 (30)	8698.4	118	4.0	14	24.7	0	62.2	0	319.5	0	309	162	191	62
bb11 (32)	6031.5	113	5.0	30	27.1	11	73.2	9	336.0	9	311	175	173	53
bb12 (32)	21133.0	110	5.0	20	32.6	0	94.8	0	557.0	0	296	169	172	56
bb13 (33)	5054.0	125	5.0	18	37.3	0	75.6	0	350.8	0	349	179	198	58
bb14 (33)	4983.8	125	5.1	17	35.8	0	75.2	0	345.3	0	349	179	203	62
bb15 (40)	—	—	12.1	—	121.3	—	374.5	—	2353.0	—	398	—	270	—
bb16 (41)	—	—	13.2	—	161.2	—	511.2	—	3506.5	—	418	—	270	—
bb17 (44)	—	—	10.4	—	126.9	—	369.3	—	2240.5	—	365	—	263	—

4 Future work

In principle, our framework can be extended to almost any power model, although this may affect the performance of our algorithm. For instance, we plan to study the effect of register assignment on energy consumption. In that case, we need to solve, for each partial solution, a register assignment problem for a partial interference graph and a partial register flow graph. Such algorithms exist in the literature [3, 8] and could be adapted for our purposes.

Currently we are working on a generalization of our method for global code generation, specifically software pipelining. This is important because considering individual basic blocks can result in worse performance than heuristic approaches that consider global and loop optimizations. In acyclic regions of the control flow graph, basic blocks with multiple entries require merging of profiles over the ingoing edges, which may lead to a loss of precision. For loops, this would additionally require a fixpoint iteration. Loop unrolling may enlarge the scope of local code generation. However, other code generation techniques for loops, such as software pipelining [21], should also be taken into account. For instance, Vegdahl [20] copies the basic block of the loop body several times and connects them with loop-carried data dependence edges. We may apply our dynamic programming algorithm on such a transformed DAG for finding optimized code. An interesting feature of our algorithm is that it allows pattern matching along loop-carried edges and hence cover nodes belonging to subsequent iterations.

5 Related work

Lee, Lee et al. [9] focus on minimizing Hamming distances of subsequent instruction words in VLIW processors. They show that their formulation of power-optimal instruction scheduling for basic blocks is NP-hard, and give a heuristic scheduling algorithm that is based on critical-path scheduling. They also show that for special multi-issue VLIW architectures with multiple slots of

the same type, the problem of selecting the right slot *within* the same long instruction word can be expressed as a maximum-weight bipartite matching problem in a bipartite graph whose edges are weighted by negated Hamming distances between microinstructions of two subsequent long instruction words.

Lee, Tiwari et al. [10] exploit the fact that for a certain 2-issue Fujitsu DSP processor, a time-optimal target schedule is actually power-optimal as well, as there the unit activation / deactivation overhead is negligible compared to the base power dissipation per cycle. They propose a heuristic scheduling method that uses two separate phases, greedy compaction for time minimization followed by list scheduling to minimize inter-instruction power dissipation costs. They also exploit operand swapping for commutative operations (multiplication).

Toburen et al. [19] propose a list scheduling heuristic that could be used in instruction dispatchers for superscalar processors such as the DEC Alpha processors. The time behavior and power dissipation of each functional unit is looked up in an ADML-like description of the processor. The list scheduler uses a dependence level criterion to optimize for execution time. Microinstructions are added to the current long instruction word unless a user-specified power threshold is exceeded. In that case, the algorithm proceeds to the next cycle with a fresh power budget.

Su et al. [17] focus on switching costs and propose a postpass scheduling framework that breaks up code generation into subsequent phases and mixes them with assembling. First, tentative code is generated with register allocation followed by pre-assembling. The resulting assembler code contains already information about jump targets, symbol table indices etc., thus a major part of the bit pattern of the final instructions is known. This is used as input to a power-aware postpass scheduler, which is a modified list scheduling heuristic that greedily picks that instruction from the zero-indegree set that currently results in the least contribution to power dissipation. The reordered code is finally completed with a post-assembler.

Work related to our general dynamic approach to integrated code generation in the context of *time* optimization is summarized in our previous work [5, 6].

6 Conclusion

We presented a framework for energy-optimal integrated local code generation. For our power model, which is generic and largely follows the standard power models in the literature, we defined a suitable power profile, which is the key to considerable compression of the solution space in our dynamic programming algorithm.

Our algorithms integrate the subproblems instruction selection, instruction scheduling and register allocation into a single optimization framework. Our method is generic and not limited to a fixed power model. If more influence factors are to be considered that are known at compile time, it can easily be adapted by modifying the power profile definition accordingly, even though ADML may need to be extended if additional parameters should be allowed in the architecture specification, but this requires only minor changes in our framework.

Acknowledgements We thank Frank Mueller and Carl von Platen for inspiring discussions.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. 27th Int. Symp. Computer Architecture*, pages 83–94, June 2000.

[2] D. M. Brooks et al. Power-aware microarchitecture. *IEEE Micro*, pages 26–44, Nov-Dec 2000.

[3] J.-M. Chang and M. Pedram. Register allocation and binding for low power. In *Proc. 32nd Design Automation Conf.* ACM Press, Jan. 1995.

[4] H.-C. Chou and C.-P. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Trans. Parallel and Distrib. Syst.*, 6(3):303–313, 1995.

[5] C. Kessler and A. Bednarski. A dynamic programming approach to optimal integrated code generation. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2001)*. ACM Press, June 2001.

[6] C. Kessler and A. Bednarski. Optimal integrated code generation for clustered VLIW architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2002)*. ACM Press, June 2002.

[7] C. W. Keßler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, Sept. 1998.

[8] E. Kursun, A. Srivastava, S. O. Memik, and M. Sarrafzadeh. Early evaluation techniques for low power binding. In *Proc. Int. Symposium on Low power electronics and design*. ACM Press, Aug. 2002.

[9] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *Proc. 13th Int. Symposium on System Synthesis*, pages 55–60. ACM Press, 2000.

[10] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and low-power scheduling techniques for embedded dsp software. In *Proc. 8th Int. Symp. on System Synthesis*, pages 110–115, 1995.

[11] S. Lee, A. Ermedahl, and S. L. Min. An accurate instruction-level energy consumption model for embedded risc processors. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 1–10. ACM Press, 2001.

[12] P. Marwedel, S. Steinke, and L. Wehmeyer. Compilation techniques for energy-, code-size-, and run-time-efficient embedded software. In *Proc. Int. Workshop on Advanced Compiler Techniques for High Performance and Embedded Processors (IWACT)*, 2001.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] M. Pedram. Power optimization and management in embedded systems. In *Proc. Asia South Pacific Design Automation Conference*. ACM Press, Jan. 2001.

[15] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sept. 2001.

[16] S. Steinke, R. Schwarz, L. Wehmeyer, and P. Marwedel. Low power code generation for a risc processor by register pipelining. Technical Report 754, University of Dortmund, Dept. of CS XII, 2001.

[17] C.-L. Su, C.-Y. Tsui, and A. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Proc. Compcon Spring '94, Digest of Papers*, pages 489–498, Feb. 1994.

[18] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. VLSI Syst.*, 2, Dec. 1994.

[19] M. Toburen, T. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *Proc. Power Driven Micro-architecture Workshop in conjunction with ISCA'98*, June 1998.

[20] S. R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*, pages 180–188. IEEE Computer Society Press, 1992.

[21] H. Yang, R. Govindarajan, G. R. Gao, and G. Cai. Maximizing pipelined functional units usage for minimum power software pipelining. Technical Report CAPSL Technical Memo 41, Dept. Electrical and Computer Engineering, University of Delaware, Sept. 2001.

[22] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: a cycle-accurate energy estimation tool. In *Proc. 37th Design Automation Conf.*, 2000.