A Performance Evaluator for Parameterized ASIC Architectures †

Jie Gong, Daniel D. Gajski and Alex Nicolau
Department of Information and Computer Science
University of California, Irvine, CA, 92717, USA

Abstract

System-level partitioning assigns functional objects such as tasks or code segments to system-level components such as off-the-shelf processors or application-specific architectures in order to meet design constraints. Characterization of these system-level components and performance evaluation of given applications on these components are crucial to system-level partitioning. In this paper, we propose a new parameterized architecture model of system-level components and present an algorithm for rapid performance estimation. The model, different from those proposed previously, reflects comprehensive architectural characteristics affecting machine parallelism. By using an ultra-fine-grain scheduler which exploits the parallelism in the model, we can evaluate performance of applications assigned to various architectures.

1 Introduction

Performance evaluation is indispensable in system partitioning [1, 2] which assigns functional objects such as tasks or code segments to system-level components such as off-the-shelf processors or application-specific architectures to satisfy system constraints. Research in [3, 4, 5] has addressed issues on performance evaluation for off-the-shelf processors while [3, 6, 7, 8] have reported methods used in performance evaluation for application-specific architectures. However, previous efforts on performance evaluation for application-specific architectures are very restrictive in the sense that they only take into account the number and type of functional units used in the architecture. They do not consider other important architectural characteristics such as the number of buses, the number of memory ports, and connection styles, which affect machine parallelism greatly.

In this work, we propose a more detailed and comprehensive parameterized architecture model which allows a designer to configure different architectures containing various functional units, storage elements, and interconnect units. Furthermore, an ultra-fine-grain scheduler incorporating various architecture styles and constraints has been developed to exploit the machine parallelism in architectures instantiated from the model. While the functionality of the ultra-fine operations is similar to microcode, we use it for purposes of performance evaluation rather than code generation. Thus, to avoid confusion, we coined a new term. The ultra-fine-grain scheduler, together with the parameterized model, enables designers to evaluate application-specific architectures more extensively and precisely.

Performance is not only affected by machine parallelism in architectures but also by program parallelism in applications. Program parallelism is a measure of the average number of operations executable concurrently given enough resources while machine parallelism is a measure of the ability of an architecture to take advantage of program parallelism. In our system, a parallelizing compiler is used to exploit massive program parallelism in applications. The role of program parallelism can be examined by scheduling an application on an architecture with and without parallelism extraction.

Our performance evaluator allows designers to evaluate different architectures and define suitable architectures for given applications. A suitable architecture is one in which machine parallelism does not limit the execution of program parallelism and at the same time machine parallelism is fully utilized. The constituents of each selected architecture can be passed on to high-level synthesis tools as allocation information to guide the synthesis of the corresponding ASIC.

An overview of the system is shown in the next section followed by an introduction to the parameterized architecture model in section 3. Algorithms used in the ultra-fine-grain scheduler are discussed in section 4. Section 5 discusses some of the experiments conducted to evaluate various architectures. Finally, section 6 describes the conclusion and the future work.

2 System overview

A block diagram of the system is shown in Figure 1. The GNU C compiler translates a C program into three-address instructions. The three-address instruction format is based on a RISC-like load/store architecture in which only load and store operations can access memory and all other operations work on registers. The compiler uses 32 registers. A fine-grain VLIW compiler with percolation scheduling and loop pipelining using unlimited resources is applied to find program parallelism beyond basic blocks and conditional boundaries [6, 9]. We have developed an ultra-fine-grain scheduler to exploit machine parallelism in architectures instantiated from the parameterized architecture model. A bypass is provided so that an application can be scheduled onto the target architecture with or without going through the VLIW compiler. We use a simulator to mimic the execution of the serial three-address code, the parallel three-address code as well as the parallel control code. The simulator records dynamic statistics such as the number of control steps executed, utilization of each resource, etc.

†This work was supported by the Semiconductor Research Corporation (grant #92-DJ-146).

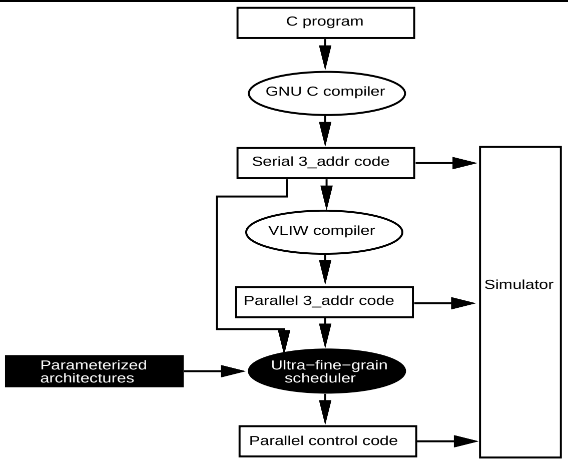

Figure 1: The overall system

In this paper, we focus on the parameterized architecture model and the ultra-fine-grain scheduler. The brevity of our discussion on other parts of the system does not imply that they are trivial or simple; their discussion is out of the scope of this paper. For details on these parts, see references [6, 9].

3 The architecture model

System-level architectures usually consist of three types of components: functional units such as adders and multipliers, storage elements such as registers and memories, and interconnect units such as selectors and buses. The parameterized architecture model defined below incorporates many architectural aspects thus enabling designers to specify a wide variety of architectures ranging from a completely serial architecture to a massively parallel architecture.

The parameterized architecture model is defined as follows:

- There are n_i functional units of type T_i (e.g. T_i could be adder, ALU, multiplier etc.), for a total of $n = \sum_{i=1}^{k} n_i$ units.
- Each functional unit is associated with a latency denoted by l_i, $l_i > 0$. Latency is defined as the delay in control steps from the input ports to the output port of a functional unit. The various features of a functional unit are specified by setting boolean flags. A true value to the *WITHLATCH* flag denotes that the corresponding functional unit has input and output latches; a true value to the *PIPELINE* flag specifies that the unit is pipelined, and the *BYPASS* flag is used to signify the presence/absence of a bypass route around a functional unit's output latch.
- There is one register file with p ports and r registers, $p > 0$ and $r > 0$. Reads from and writes to the register file are assumed to take 1 control step.
- There is one memory module with m ports, $m > 0$, and a latency of l, $l > 0$. A port has one *MAR* and one *MBR* latch associated with it.
- There are b buses used to connect different components, $b > 0$. Interconnections are specified by a binary connection table M. The row indices of the matrix M are buses and the column indices of M are latches or ports. Entry $M[i,j]$ has value '1' if there is a connection between bus_i and $element_j$, otherwise $M[i,j]$ has value '0'.

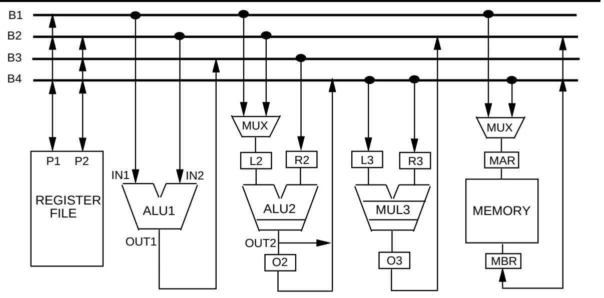

Figure 2: A concrete architecture

A concrete architecture can be obtained by specifying values for the parameters. Figure 2 shows an architecture consisting of two ALUs and one multiplier. The latencies of *ALU1*, *ALU2*, and *MUL3* are 1, 2, and 3 respectively. *ALU1* has no input/output latches. *ALU2* and *MUL3* have left, right, and output latches: $L2$, $R2$, $O2$, $L3$, $R3$, and $O3$. *ALU2* has a bypass route around its output latch. *IN1*, *IN2*, and *OUT1* denote the input/output ports of *ALU1*. There is a register file which has two ports, $P1$ and $P2$, and one memory module which has one port with two latches, *MAR* and *MBR*. There are four buses, $B1$, $B2$, $B3$ and $B4$, as well as some multiplexers for connections among these different components. The interconnection table for this concrete architecture is shown in Figure 3.

	IN1	IN2	OUT1	L2	R2	O2	OUT2	L3	R3	O3	P1	P2	MAR1	MBR1
B1	1	0	0	1	0	0	0	0	0	0	1	0	1	0
B2	0	1	0	1	0	0	0	0	0	1	1	1	0	1
B3	0	0	1	0	1	0	0	0	0	0	0	1	0	0
B4	0	0	0	0	0	1	1	1	1	0	1	1	1	1

Figure 3: Connection table for architecture in Figure 2

4 The ultra-fine-grain scheduler

The ultra-fine-grain scheduler takes either serial three-address code or parallel three-address code as input and maps it onto the target architectures specified by the architecture model described in the last section. It compiles three-address operations into control code (i.e. ultra-fine operations) which describes data transfers between architectural elements. As an example, consider the execution of a three address addition "$Reg_3 = Reg_1 + Reg_2$" on *ALU2* in Figure 2. One possible sequence of the data transfers is: $Reg_1 \rightarrow L2$, $Reg_2 \rightarrow R2$; $L2\ ALU2\ R2 \rightarrow O2$; $O2 \rightarrow Reg_3$ or $Reg_1 \rightarrow L2$, $Reg_2 \rightarrow R2$; $L2\ ALU2\ R2 \rightarrow Reg_3$ if flag *BYPASS* of *ALU2* is set. Whether a set of data transfers can be scheduled in parallel is determined by the availability of required resources. For example, suppose at the control step when $Reg_1 \rightarrow L2$ is scheduled, bus $B3$ is occupied by another data transfer, then even if register file ports are available, the data transfer $Reg_2 \rightarrow R2$ has to wait until $B3$ is available. Scheduling parallel data transfers under architecture constraints is the main task of the ultra-fine-grain scheduler.

The ultra-fine-grain scheduler consists of three components: a basic block recognizer, a data dependence graph builder and a scheduler and binder. We briefly introduce the first two tasks which use standard techniques and discuss our algorithm for the third task, i.e., scheduling for a target architecture.

4.1 The basic block recognizer

The basic block recognizer takes serial or parallel three-address code and finds the basic blocks in the code. It uses a standard algorithm which basically finds the entry node of a basic block first, then lists the operations that belong to the same basic block until the entry node of the next basic block is found.

4.2 The data dependence graph builder

The dependence graph builder takes a basic block as input and produces a data dependence graph (DDG) for the list of three-address operations, L, of the basic block. A DDG is a directed acyclic graph $G = (V, E)$. Each node in V is a three-address operation. Each edge in E is one of the three types of dependence edges: (1) flow dependence (write-read) edges; (2) anti-dependence (read-write) edges; and (3) output dependence (write-write) edges. The detailed algorithm to build a DDG is reported in [10]. The algorithm creates a node in DDG for each three-address in L. Then the algorithm creates the flow dependence edges, the output dependence edges and the anti-dependence edges among the nodes in DDG.

Given a DDG that exposes program parallelism, the mobility of each node in DDG is computed. Mobility [11] is a measure of urgency that a node (operation) needs to be scheduled. A smaller value of mobility indicates a higher urgency for scheduling. Nodes in the critical path have zero mobility; they need to be scheduled as soon as they are ready and resources are available. A ready node represents an operation that has all predecessors already scheduled.

4.3 The scheduler and binder

The scheduler and binder takes a DDG as input and then schedules and binds operations in the DDG to resources given in the target architecture. The scheduling algorithm is a variation of list scheduling [11], which incorporates various architectural parameters. The algorithm shown below as Algorithm 1 basically considers the required functional units and memory resources for the operations first and then considers the required bus and register file port resources. Next, we describe each procedure used in the algorithm.

Assume the target architecture has n_i functional units of type T_i, i = 1, ..., k. The algorithm uses a priority list L for each type T_i to queue corresponding ready nodes. These lists are denoted by the variables $L_{T_1}, \ldots, L_{T_k}$. The algorithm also builds a priority list L_M to queue memory type operations such as store or load operations. Besides these priority lists, the algorithm uses a special priority list L_W to queue the operations which have been assigned to either functional units or memory ports but are waiting for bus or register file port resources. L_W is a list of tuples $< op, r >$ indicating that operation op has been assigned to the resource r. Each priority list is sorted with respect to a priority function. In the algorithm, the mobility of a node is used as the priority function. Nodes are sorted in ascending order of their mobilities. Initially all priority lists are empty.

Algorithm 1: Scheduling DDG to Target Architecture

```
C = 0;
INSERT_RDY_OPS(DDG, $L_{T_1}, \ldots, L_{T_k}, L_M$, C+1);
while $(L_{T_1} \cup \ldots \cup L_{T_k} \cup L_M \cup L_W \cup DDG) \neq \phi$ do
    C = C + 1;
    /* assign functional units to their ready lists */
    for i = 1 to k do
        while $(Q_{T_i} \neq \phi) \cap (L_{T_i} \neq \phi)$ do
            r = FIRST($Q_{T_i}$);
            op = FIRST($L_{T_i}$);
            INSERT($L_W$, $< op, r >$);
            $Q_{T_i}$ = DELETE($Q_{T_i}$, r);
            $L_{T_i}$ = DELETE($L_{T_i}$, op);
        end while
    end for
    /* assign memory resource to its ready list */
    while $(Q_M \neq \phi) \cap (L_M \neq \phi)$ do
        r = FIRST($Q_M$);
        op = FIRST($L_M$);
        INSERT($L_W$, $< op, r >$);
        $Q_M$ = DELETE($Q_M$, r);
        $L_M$ = DELETE($L_M$, op);
    end while
    /* assign bus and register file port resource to $L_W$*/
    for each $< op, r > \in L_W$ do
        if CONNECT_RESERVED(op, r, C, CRT)
        then
            SCHEDULE_OP(op, r, C);
            x = RESOURCE_RELEASE_STEP(r, C);
            RRQ = INSERT(RRQ, $< r, x >$);
            for each outgoing edge e of op do
                y = EDGE_DELETION_STEP(op, e, C);
                EDQ = INSERT(EDQ, $< e, y >$);
            end for
            $L_W$ = DELETE($L_W$, $< op, r >$);
        end if
    end for
    /* recollect resources released at next step */
    RECOLLECT_RESOURCES(RRQ, $Q_{T_1}, \ldots, Q_{T_k}$,
                        $Q_M$, C+1);
    /* delete edges needed to remove at next step */
    DELETE_EDGE(EDQ, DDG, C+1);
    /* insert nodes ready at next step in lists */
    INSERT_RDY_OPS(DDG, $L_{T_1}, \ldots, L_{T_k}, L_M$, C+1)
end while
```

There is a queue for each type of resource. These queues are denoted by the variables $Q_{T_1}, \ldots, Q_{T_k}$, and Q_M. Initially there are n_i resources of type T_i in Q_{T_i}. Suppose two ALUs of the same type and one multiplier exist in the given architecture. Then, initially, there are two resources *ALU1*, *ALU2* in Q_{ALU} and one resource *MUL3* in Q_{MUL}. If the memory has k ports, then there are k resources in Q_M initially. Recall that the connection of the architecture is represented by a matrix M. The row indices of the matrix are buses. The column indices of the matrix are latches and ports. Entry $M[i, j]$ has value '1' if there is a connection between bus_i and $element_j$, otherwise $M[i, j]$ has value '0'. Note that a register file port or a bus is always used by only one control step and can always be released in the next control step, therefore, we do not use resource queues for register file ports or buses. Instead, we use a connection-reservation table (CRT) to record whether they are used at certain control steps. The row indices of CRT are buses and register file ports. The column indices of CRT are control steps. $CRT[i, j]$ is '1' if $element_i$ is reserved at control step j, otherwise it is '0'. Initially, all entries in CRT are zeros.

There are two auxiliary data structures. A resource-releasing queue (RRQ) is used to record at which control step C the occupied resources r can be released. It is a queue of tuples $< r, C >$. An edge-deletion queue (EDQ) is used to record at which control step C the edge e can be deleted from the DDG. It is a queue of tuples $< e, C >$ Initially, RRQ and EDQ are empty.

INSERT_RDY_OPS$(DDG, L_{T_1}, \ldots, L_{T_k}, L_M, C)$ inserts nodes in DDG, that are ready at control step C, into their corresponding priority lists. A ready node in DDG is one whose in-degree is zero i.e., has no incoming edges. *INSERT_READY_OPS* deletes those ready nodes from DDG but does not delete their outgoing edges.

DELETE(L, x) returns the new L in which x has been removed. *INSERT*(L, x) returns the new L in which x has been inserted. *FIRST*(L) returns the first element of L.

CONNECT_RESERVED(op, r, C, CRT) returns false if the bus and register file port resources required by op can not be fulfilled in CRT. Otherwise, *CONNECT_RESERVED* returns $TRUE$ and marks those entries of CRT reserved by op. The reservation process uses a look-ahead approach. For example, let's assume that the current control step is C, and resource r to which operation op is assigned is a functional unit with latency of l and has both input and output latches. The reservation procedure will try to look up column C and column $C+l+1$ of CRT as well as connection matrix M to decide if there are buses and register file ports available for scheduling the operation op, since op will need buses and register file ports at control step C and control step $C + l + 1$ to move data between register file and latches. If the resources are available, then operation op will be scheduled at this control step. Otherwise it has to be delayed.

SCHEDULE_OP(op, r, C) creates the scheduling sequence for op starting from control step C. The sequence depends on the resource type, latency, and style etc. For example, suppose r is a functional unit with latency of 1 and has both input and output latches. The scheduling sequence for op will be, (1) in control step C, contents of source registers are moved to input latches of r, (2) in step $C + 1$, 'L_r op R_r' are moved to O_r where L_r, R_r and O_r are the left, right, output latches of r respectively, (3) In step $C + 2$, contents of the output latch is moved to the destination register.

RESOURCE_RELEASE_STEP(r, C) returns the control step at which the occupied resource r is released. This step depends on the resource type, latency, and style (e.g. pipelined/non-pipelined) as well as the current control step C. For example, suppose r is a functional unit with latency l and it is used at step C, r will be available at step $C+1$ only if it is pipelined. If r is not pipelined, it will be available at step $C + l$.

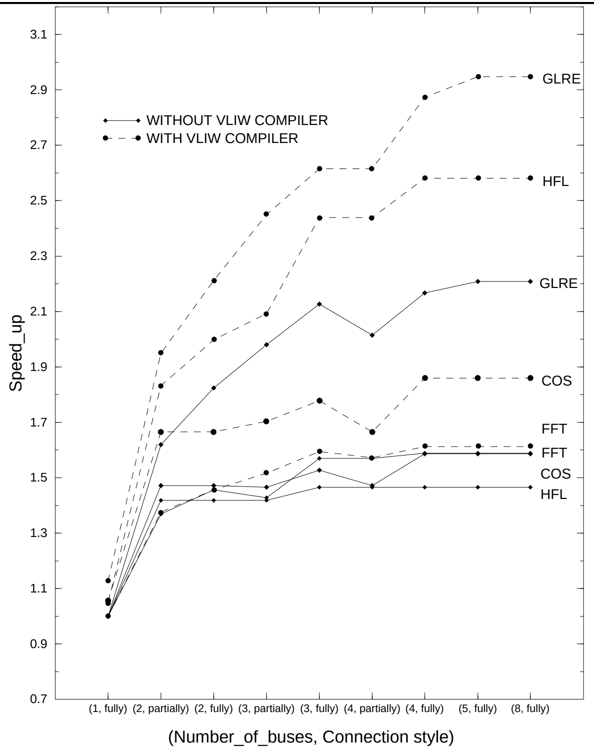

Figure 4: A study for bus constraints

EDGE_DELETION_STEP(op, e, C) returns the control step at which an outgoing edge e of op in DDG should be removed. This step depends on the resource type, latency, style, current control step C as well as the type of the edge. For example, a outgoing flow dependence edge of op can be deleted after the step at which the destination register of op is written while an anti-dependence edge can be deleted after the source registers of op are read.

RECOLLECT_RESOURCES$(RRQ, Q_{T_1}, \ldots, Q_{T_k}, Q_M, C)$ finds out from RRQ all resources released at the control step C and inserts them to their corresponding resource queues.

DELETE_EDGE(EDQ, DDG, C) finds out from EDQ all edges that need to be deleted from DDG at control step C and deletes them from DDG.

5 Experimental results

To evaluate different architectures with various constraints several orthogonal experiments have been conducted. Four applications are used in the experiments: a fast Fourier transformation program (FFT), a Cosine computation program (COS), a hydro fragment loop (HFL) and a program of general linear recurrence equations (GLRE). The absolute performance of a program is measured by the number of control steps required to execute the code. Speed_up is used to measure relative performance and defined as x/y, where x is the performance obtained for the reference architecture whereas y is the performance obtained for the architecture to be evaluated.

5.1 Bus constraints

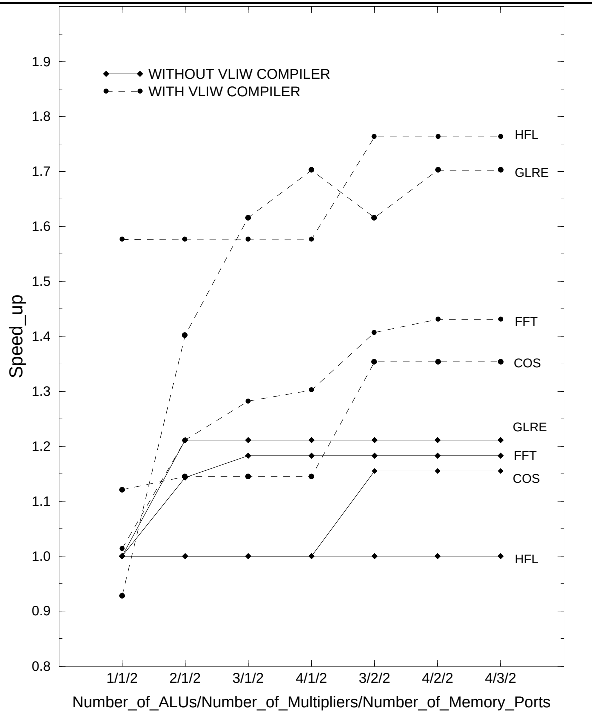

Figure 5: A study for functional unit constraints

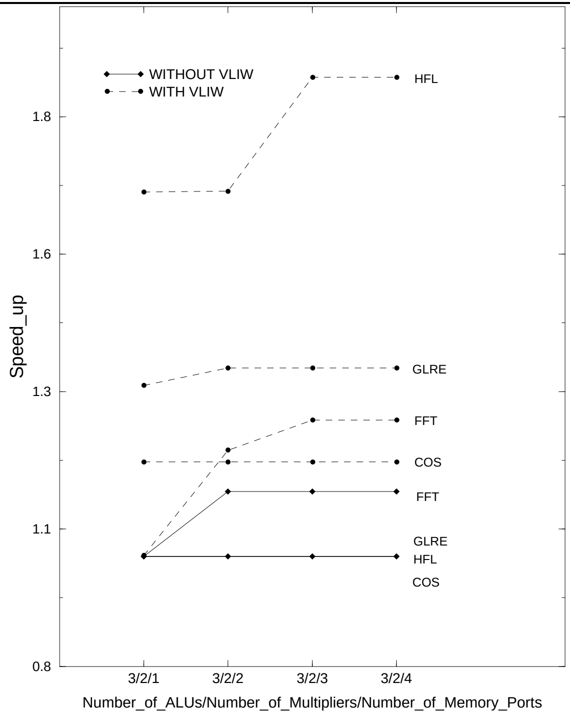

Figure 6: A study for memory port constraints

We first study the impact of bus constraints on performance. The chosen architecture has one ALU with a latency of 1 and one Multiplier with a latency of 3. Both units are pipelined and have input and output latches. The architecture has a one-port memory with a latency of 2.

We use different numbers of buses and connection styles for the chosen architecture. Functional units and memory are fully or partially connected to the buses. The register file in the architecture has the same number of ports as the number of buses used. Register file ports are fully connected to the buses, i.e., every register file port has a connection to a bus.

The results are shown in Figure 4. The vertical axis represents performance speed up compared with a reference point. The reference point in the graph is the performance of the program without going through the VLIW compiler and executing on the architecture with 1 bus. The horizontal axis represents different architecture constraints. The first element of the tuple indicates the number of buses. The second element of the tuple indicates bus connection styles. *Fully* denotes the fully-connected style while *partially* denotes a partially-connected style. Details of the partially-connected style for each architecture can be found in [10].

The results show that, increasing bus number from 1 and 2 can improve the performance for all applications dramatically. For some applications with moderate parallelism, 3 buses will be enough. For applications with lots of parallelism, 4 buses would be good. We can view these programs after parallelism extraction by the VLIW compiler as new programs with more parallelism. The results also show that unless resources are highly constrained, using the parallelism beyond basic blocks (extracted by the VLIW compiler) can dramatically improve performance.

5.2 Functional unit constraints

To study the impact of the number of functional units on performance, we assume an architecture that has ALUs with a latency of 1, multipliers with a latency of 3, one memory with a latency of 2 and 2 ports. Every unit has input-output latches. The ALUs and multipliers are not pipelined. There are 4 buses which are fully connected with units and ports. There is a register file with 4 ports.

The results are shown in Figure 5. For some applications, the performance improves along with the increment in the number of ALUs. For other applications, increasing the number of multipliers improves the performance. This shows that for different types of applications, different types of functional units should be provided in order to improve performance.

One interesting phenomenon is that program GLRE after VLIW parallelizing compiler performs worse than that without using VLIW parallelizing compiler on the architecture with 1 ALU and 1 multiplier. The reason is that the version of the VLIW compiler used in these experiments introduces some overahead such as duplicated operations in different blocks during the parallelization process. The small amount of machine parallelism offered in the architecture can not compensate for the overhead introduced.

5.3 Memory port constraints

In this experiment, we study how the number of memory ports affects performance. The architecture we use has 2 multipliers with a latency of 3, 3 ALUs with a latency of 1, 1 memory with a latency of 2. Every unit has input-output latches. ALUs and multipliers are not pipelined. There are 3 buses that are fully connected with functional

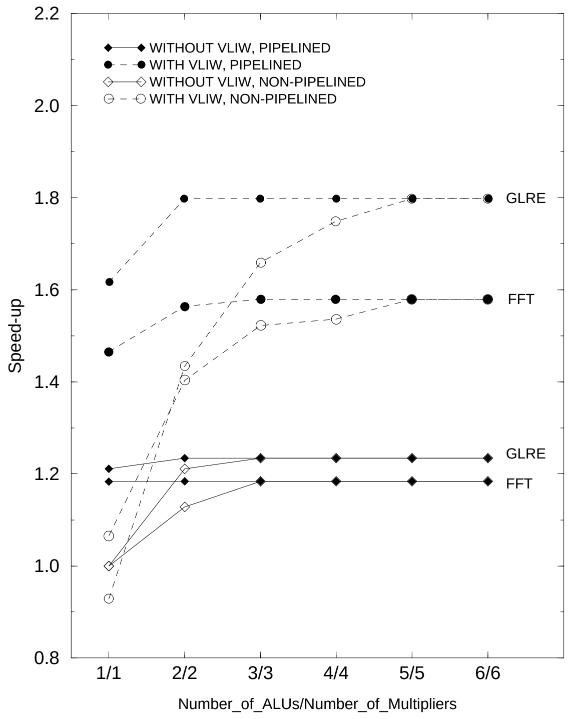

Figure 7: A study for pipeline vs. nonpipeline

units and memory/register file ports. There is a register file with 3 ports.

Figure 6 shows the results. We notice that increasing the number of memory ports helps applications which have more memory accesses such as the FFT. And it does not help performance for applications with few memory accesses. While not surprising, this underlines the need for a tool such as the one proposed to enable study of application characteristics and their interaction with architecture selection in ASIC design.

5.4 Temporal parallelism vs. spatial parallelism

In this experiment we compare two types of parallelism: temporal parallelism which is achieved by pipelining and spatial parallelism which is achieved by using multiple functional units. The architecture we use has 6 buses, fully connected to functional units and memory/register file ports. There is a memory with 4 ports and a register file with 6 ports. ALU has a latency of 1, multiplier has a latency of 3 and memory has a latency of 2. Every unit has input-output latches.

We observe from Figure 7 that a design with 1 pipelined ALU and 1 pipelined multiplier results in machine parallelism similar to a design with 2 or 3 non-pipelined ALUs and 2 or 3 non-pipelined multipliers. Essentially, pipelining requires less resources to achieve similar performance.

6 Conclusion and future work

We have presented a comprehensive parameterized model to characterize various architectural styles and constraints and an ultra-fine-grain compiler to exploit machine parallelism in architectures instantiated from the architecture model. A set of experiments have been conducted to show how various architectures can be evaluated. Through such an evaluation, architectures suitable for given applications can be selected. The constituents of each selected architecture can be passed on to high-level synthesis tools as allocation information to aid the synthesis of the corresponding ASIC.

Future work will be on conducting more experiments to study the the performance impact of other architectural parameters such as the number of register file ports and the number of pipeline stages of functional units as well as the combined impact of various architectural parameters. Also, we will explore the possibility of extending the parameterized architecture model from bus-based model to point-to-point model.

The algorithm used in the ultra-fine-grain scheduler has room for improvement. For example, heuristics besides mobility can be used in the priority function to exploit more parallelism. Also, the use of a more sophisticated VLIW compiler that takes bus constraints and operation pipelining into account may further improve the results.

7 Acknowledgements

Our system is built on a version of the VLIW compiler implemented by the PS group at UC Irvine. We would like to thank Roni Potasman for his discussions and suggestions regarding this work.

References

[1] R. Ernst and J. Henkel, "Hardware-software codesign of embedded controllers based on hardware extraction," in *International Workshop on Hardware-Software Co-Design*, 1992.

[2] R. Gupta and G. DeMicheli, "System-level synthesis using re-programmable components," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 2–7, 1992.

[3] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proceedings of the International Conference on Computer Design*, pp. 452–457, 1993.

[4] J. Gong, D. Gajski, and S. Narayan, "Software estimation from executable specifications," in *Journal of Computer and Software Engineering*, 1994.

[5] W. Wolf and J. Martinez, "C program performance estimation for embedded systems architecture sizing," in *International Workshop on Hardware-Software Co-Design*, 1993.

[6] R. Potasman, *Percolation-based compiling for evaluation of parallelism and hardware design trade-offs.* PhD thesis, University of California, Irvine, 1992.

[7] S. Narayan and D. Gajski, "Area and performance estimation from system-level specifications." UC Irvine, Dept. of ICS, Technical Report 92-16,1992.

[8] A. Timmer, M. Heijligers, and J. Jess, "Fast system-level area-delay curve prediction," in *Proceedings of APHDLSA*, 1993.

[9] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," in *Proceedings of International Conference on Parallel Processing*, 1985.

[10] J. Gong and D. Gajski, "Exploiting ultra-fine grain parallelism for machines with parallel pipelined datapaths." UC Irvine, Dept. of ICS, Technical Report 92-112,1992.

[11] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Boston, Massachusetts: Kluwer Academic Publishers, 1991.