Transactions Briefs

RTGEN—An Algorithm for Automatic Generation of Reservation Tables From Architectural Descriptions

Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau

***Abstract*—Reservation Tables (RTs) have long been used to detect conflicts between operations that simultaneously access the same architectural resource. Traditionally, these RTs have been specified explicitly by the designer. However, the increasing complexity of modern processors makes the manual specification of RTs cumbersome and error prone. Furthermore, manual specification of such conflict information is infeasible for supporting rapid architectural exploration. In this paper, we present an algorithm to automatically generate RTs from a high-level processor description with the goal of avoiding manual specification of RTs, resulting in more concise architectural specifications and also supporting faster turnaround time in design space exploration. We demonstrate the utility of our approach on a set of experiments using the TI C6201 very long instruction word digital signal processor and DLX processor architectures, and a suite of multimedia and scientific applications.**

I. Introduction

In most modern processors that exhibit multiple levels of parallelism and deep pipelines, resource and data hazards can lead to significant performance degradation or even [for very long instruction words (VLIWs)] incorrect execution behavior. Thus, detection and avoidance of such hazards is a crucial task in processor-based system design. Hazard information may be captured as conflicts between operations that access the same resources at the same time. Reservation tables (RTs), which specify (and represent) both the pipeline behavior and resource usage of operations, are commonly used as part of the machine model for capturing such conflict information for retargetable compilers. RTs can be used, for example, by the instruction scheduler to avoid resource conflicts and pipeline hazards. Complex processors are increasingly being deployed in high-end embedded applications, typically as (fixed or parameterized) cores in a system-on-chip (SOC). Since RTs can be specified at different levels of detail, they can also be used in an architectural design space exploration (DSE) environment involving tradeoffs between accuracy and speed of the software tools for embedded SOCs.

Most current retargetable tools that follow the RT's approach require the user to specify the RTs manually on a per-operation basis in the architecture description language (ADL). Processors that contain complex pipelines, large amounts of parallelism, and complex storage sub-systems, typically contain a large number of operations[1] and resources (and, hence, RTs). Manual specification of RTs on a per-operation basis thus becomes cumbersome and error prone. Furthermore, exploration and customization of different architectures drives the need for rapid evaluation of different architectural (and pipeline) configurations—making it impractical to manually specify RTs on a per-operation basis for each configuration.

In this paper, we present an RT generation (RTGEN) algorithm, which automatically generates RTs from a high-level processor description. This frees the user from the burden of having to manually enumerate the RTs, allowing for conciseness of specification, reduction of errors in specification, and reduction of time spent in specification.

Moreover, in the context of VLIW processors, a complete set of RTs is necessary for correct scheduling of the operations. Incorrect or incomplete RTs (e.g., containing an optimistic resource usage of operations or missing an important resource), may lead to incorrect scheduling and execution of the program. For instance, in the TI C6x processor, missing the resource describing the cross-register file-path connection may lead to scheduling in the same VLIW instruction of multiple operations, which try to use the same connection at the same time, resulting in a resource hazard.

Furthermore, in the absence of an RT model (assuming a non-pipelined version of the compiler) or using coarse grain RTs, such as optimistic RTs (leading to aggressive compilation), or pessimistic RTs (leading to conservative compilation), the performance of the generated code may be significantly impacted.

Our compiler uses Trailblazing percolation scheduling (TiPS) [18] to schedule the generated RTs in the pipeline. TiPS is a powerful instruction-level parallelism technique that parallelizes operations across basic blocks. Our RTGEN approach is applicable to a diverse set of architectures (such as VLIW, Superscalar, and RISC). We described and successfully generated a retargetable compiler and simulator for the TI C6201 VLIW digital signal processor (DSP), and the Motorola PowerPC Superscalar. For the Motorola PowerPC, we compared the code generated by our retargetable compiler using RTs to the native Motorola compiler, obtaining similar results.

Since every operation proceeds through a pipeline path and accesses storage units through some data-transfer paths, the key idea behind the RTGEN approach is that it is possible to trace the execution of the operation through the architectures pipeline and data-transfer segments and, thus, generate accurate RTs.

In Section II, we describe related work on ADL-driven pipeline (and constraint) specification for tools and compare them with our approach. In Section III, we motivate the need for automatic RTGEN using the TI C6201 VLIW DSP. Section IV describes the features necessary in a high-level processor description to support automatic generation of RTs. We use EXPRESSION, an ADL designed to support architecture exploration and software tool-kit generation. The RTs generated from EXPRESSION are used to drive the Trailblazing scheduler in EXPRESS, a highly optimizing memory-aware instruction-level parallelizing (ILP) compiler. Section V presents the algorithm for automatic RTGEN. Section VI presents experiments using an implementation of the RTGEN algorithm, conducted to demonstrate utility of this approach, and a brief discussion on the different usage scenarios of our approach for the purpose of DSE, while Section VII concludes this paper.

Manuscript received September 19, 2000; revised September 5, 2001. This work was supported in part by the National Science Foundation under Grant MIP-9708067, by the Defense Advanced Research Projects Agency under Grant F33615-00-C-1632, and by Motorola under two fellowships.

The authors are with the Architectures and Compilers for Embedded Systems (ACES) Laboratory, Center for Embedded Computer Systems, University of California at Irvine, Irvine, CA 92697-3425 USA (e-mail: pgrun@cecs.uci.edu; ahalambi@cecs.uci.edu; dutt@cecs.uci.edu; nicolau@cecs.uci.edu).

Digital Object Identifier 10.1109/TVLSI.2003.813011

[1]For single-issue machines, the terms operation and instruction are used interchangeably. For multiissue machines (e.g., VLIW and Superscalar), an instruction represents a set of operations issued/executed simultaneously.

II. Related Work

While pipelining was first developed during the late 1950s, most modern pipelining techniques are direct descendents of work done during the late 1970s and early 1980s. Reference [15] surveys pipelining techniques and provides most of the terminology and concepts in use today. Reference [13] contains a good description of the various aspects of pipelining (including tackling hazards and compilation techniques).

The advent of SOC technology, with the ability to explore between a variety of processor cores, has led to renewed interest in retargetable software tool kits (e.g., compilers and simulators). Due to increased parallelism and pipelining in today's processors, more complex data and resource hazards may raise conflicts in the architecture, leading to insertion of stalls in the pipeline, or even incorrect execution of instructions (for VLIW processors). Thus, it is crucial to detect and avoid such conflicts either in the software toolkit (e.g., compiler), or in the processor itself (in the hardware controller). We present related approaches to specifying and using such conflict information.

Traditionally, RTs are used to detect conflicts for scheduling [17]. The concept of using RTs to represent the resources used by individual operations in each stage of the pipeline was developed in [2] and [4]. Conflicts between operations are detected by comparing their RTs. Examples of compilers that adopt this approach include the multiflow trace scheduling compiler [7] and the Trimaran (Elcor) compiler [24]. Trimaran uses the MDes [6] ADL, which captures constraints between operations with explicit RTs on a per-operation basis, using a hierarchical description for compactness. However, explicit specification of RTs introduces redundancy in the processor description. Moreover, during DSE structural changes to the architecture may propagate through the description, also requiring the user to manually reflect the changes in the RT section.

Alternatively, state diagrams or finite state automatons (FSAs) have been used to represent the set of all legal instruction schedules for a processor. Since the FSAs are derived from RTs, RTs are a prerequisite in this approach. References [1] and [20] present compiler techniques that use FSAs. For determining operation conflicts, the RT approach suffers from the drawback of increased compilation time as compared to the FSAs. However, [3] and [6] present RT optimization techniques that can be used to mitigate these drawbacks. Further, RTs are needed in order to generate FSAs [15]. A disadvantage of the FSA approach is that it is not amenable to certain advanced scheduling techniques (such as iterative modulo scheduling [21] and mutation scheduling [19]). In our compiler, we use mutation scheduling together with TiPS [18] to better match the application code to the complexities of the architecture.

The LISA [8] and RADL [22] approaches are targeted mainly to generate high-performance simulators. The conflicts are modeled as signals that capture at run time the occurrence of conflicts in the pipeline stages. In the nML ADL [9], the processor's instruction-set (IS) is described as an attributed grammar with the derivations reflecting the set of legal combinations of operations. Combinations of operations not recognized by this grammar represent the conflicts. In the instruction set description language (ISDL) [5], ADL illegal combinations of operations are explicitly enumerated. While these approaches have the advantage of being able to capture most of the constraints (including those due to bit-width restrictions), the size of specification tends to get very large for complex processors. Reference [16] presents a technique for automatic extraction of the IS, from a structural description, specified in the MIMOLA ADL.

All the previous approaches presented require manual specification of the conflicts, which is a tedious and error-prone task. We present an algorithm to automatically generate the set of RTs from a mixed structural/behavioral description of the processor. We thereby free the user from the burden of having to manually specify the RTs. Moreover, during DSE, changes to the structure of the processor are reflected automatically in the RTs, allowing for fast DSE iterations. Also, by automatically generating the conflict information, we avoid redundancy in the input processor specification. In our approach, the same architectural specification is used to generate both a structural simulator SIMPRESS [14], as well as the RTs required by our optimizing compiler EXPRESS.

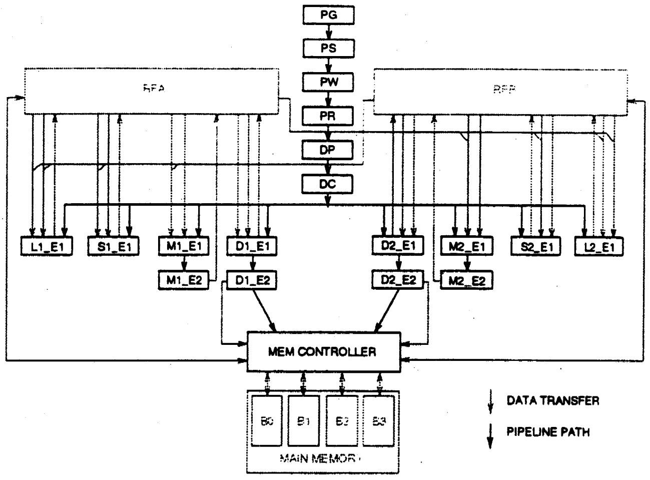

Fig. 1. Block diagram of the TI C6201 VLIW DSP. $PG, PS, PW,$ and PR perform instruction fetch. DP and DC perform decode and dispatch. $L, S, M,$ and D are functional units.

Furthermore, using accurate RTs in compilation may significantly improve the performance of the generated code by better overlapping the independent operations and better utilizing the pipeline resources. In particular, for memory operations, which tend to be very time consuming (especially for the off-chip dynamic random access memory (DRAM) accesses), better hiding their latency leads to substantial performance improvements. In [10], we present the usage of the generated timing and resource (i.e., RTs) information in the context of off-chip DRAM memories, exhibiting page- and burst-mode accesses. By providing the compiler with accurate timing and RTs for the memory operations, compared to the traditional approach using optimistic timings and RTs, we generate an average of 24% performance improvement. Moreover, in [11], we use the generated timing and RTs for scheduling the cache hit-and-miss operations. By providing the compiler with accurate timings and RTs for the cache operations, we generate an average of 61% performance improvement over the traditional approach using optimistic models.

III. Motivating Example

We use Texas Instruments Incorporated's C6201 VLIW DSP to intuitively explain the RTGEN and illustrate the complexity of the problem. The C62 is a state-of-the-art fixed-point DSP with a high-performance VLIW architecture, whose block-level diagram is shown in Fig. 1. The bold blocks represent pipeline and functional units, while the dotted blocks represent storage components. The interesting features of this architecture include ILP with up to eight operations being issued in one cycle, a complex fragmented pipeline and a complex storage subsystem with two register files (RFA, RFB) with multiple read/write ports and a main memory with four banks.

For ease of illustration, we have omitted showing the main controller, pipeline latches, ports, and some connections and storages. However, the actual RTGEN assumes a complete specification including, for example, the source and destination ports for each functional unit. A detailed description of the TI C6201 architecture can be found in [23].

The TIC6201 processor model we examined has 426 fully qualified operations,[2] requiring the specification of 426 RTs. The operation formats supported are the 1-operand *(OPCODE DEST)*, 2-operand *(OPCODE DEST SRC1)*, and 3-operand *(OPCODE DEST SRC1 SRC2)* formats.

The large number (426) of operations makes specification of RTs on a per-operation basis very tedious and error prone. Further, specifying (or generating) RTs is not a straightforward simple task for most architectures due to the presence of complex architectural features (e.g., the C62 has fragmented pipeline paths, multiple register files with cross paths, and varied operation formats). An automatic RTGEN approach is essential to free the user from the burden of specifying complex RTs and reduce the possibility of errors in the RTs. Our approach results in automatic generation of RTs even for architectures with complex features (including multiple pipeline paths, bus-based data transfers, and different operation formats).

IV. ADL Information Required for RTGEN

We now describe the essential features required in an ADL to support automatic generation of RTs. While we use our ADL EXPRESSSION as a vehicle for demonstrating the automatic generation of RTs from a machine description, it is important to note that the RTGEN approach we describe is not specific to EXPRESSION. Indeed, any ADL that incorporates the (generic) features mentioned below is a candidate for automatic generation of RTs using our approach.

The primary characteristic of an ADL for automatic RTGEN is integrated specification of both structure and behavior (i.e., IS) of the system. Below, we summarize the key features of the structural and behavioral specification of such an ADL.

ADL STRUCTURAL SPECIFICATION: The structure (of a processor system) is defined by its *components* and the *connectivity* between these components. Further, each component is defined by its attributes and the connectivity between components is defined using two high-level constructs (*pipeline* and *data transfer*), as described below.

Component Specification: Every component in the architecture may be modeled as a *unit* (e.g., ALU), a *storage* (e.g., register file), a *port* or a *connection* (e.g., bus). Each component is further described in terms of its *OPCODES* (operations supported by the component), *TIMING* (for multicycle or pipelined components), and *LABEL* (a tag associated with port/connection components, which, together with the OPCODES construct, ties the behavioral description to the structural description of components).

Connectivity Specification: The *PIPELINE* and *DATA TRANSFERS* constructs provide a natural and concise way to specify the net list at a high level. PIPELINE is used to specify the ordering of units that comprise the architecture's pipeline stages. *Pipeline paths* represent the sequence (through time) of execution for the pipeline units. DATA TRANSFERS are used to specify the valid unit-to-storage or storage-to-unit data transfers. *Data-transfer paths* typically occur between functional units (e.g., ALUs) and memory elements (e.g., register files).

ADL BEHAVIORAL SPECIFICATION: The behavior of a processor is defined by its IS. Each operation in the IS is defined in terms of its *OPCODE* (the opcode mnemonic associated with the operation), *OPERANDS* (the list of arguments—e.g., src, dst—associated with the operation), and *FORMAT* (the operation format used to indicate the relative ordering of the various operation fields).

[2]A *fully qualified operation* has all of its fields bound to architectural components such as functional and storage units.

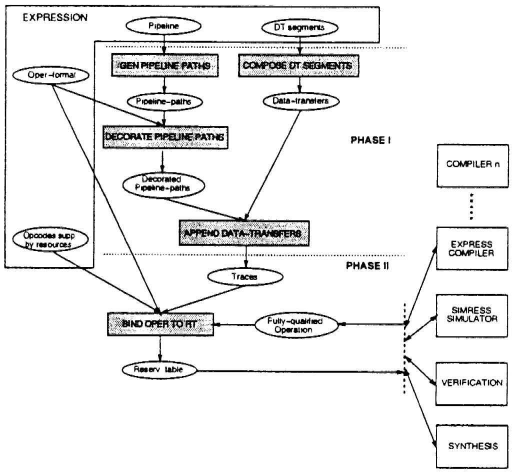

Fig. 2. Overview of the RTGEN algorithm.

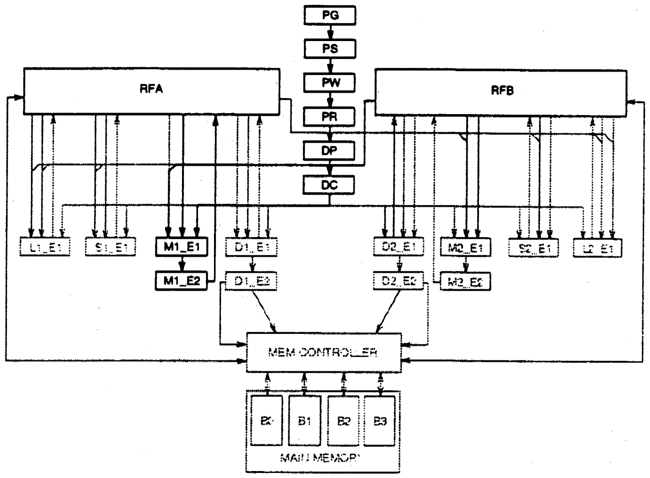

Fig. 3. Trace in the TI C6201 architecture.

The information captured in the ADL allows us to generate resource conflicts for multiissue processors (e.g., VLIW, Superscalar), with fragmented pipeline paths, heterogeneous functional units, and variable latency operations (e.g., an ADD and a MAC operation on the same functional unit may have different latencies and, similarly, an ADD operation on different functional units may have different latencies).

These ADL features are used to drive the automatic generation of RTs as described in Section V.

V. RTGEN: An Algorithm for Automatic Generation of RTs

Fig. 2 presents the flow of RTGEN. RTGEN starts from a description of the processor, specified in an ADL such as EXPRESSION, and generates the RT for a given operation. RTGEN proceeds in two phases. In the first phase, the pipeline paths and data transfer segments are combined to generate a cross-product called traces. Traces do not incorporate IS information, but instead capture the behavior of the pipeline as a set of possible execution footprints in the netlist. A trace in our example, the TI C6201, is shown in Fig. 3 with the bold lines traversing the PG, PS, PW, PR, DP, DC, M1_E1, and M1_E2 units, and accessing RFA and RFB. This trace could be activated by an MPY operation, but

```
Algorithm: Phase-I-Generate-Traces
Input: Structural specification of the processor (currently in EXPRESSION)
Output: All possible Traces in the architecture
Begin Phase I
  pipeline_paths=Get_pipeline_paths_from_ADL(expression_spec);
  data_transfers=Get_data_transfers_from_ADL(expression_spec);
  decorated_pps=Decorate_pipeline_paths(pipeline_paths);
  traces=Append_data_transfers_to_pipeline_paths(decorated_pps,
     data_transfers);
  return traces;
End Phase I

  Procedure: Decorate_pipeline_paths(pipeline_paths)
  Input: The pipeline_paths
  Output: The decorated pipeline paths
  Begin
    for each pipeline_path p
       for each operation format f
          for all permutations of resources r implementing the operands from f
             decorate p with the resources r
             add decorated pipeline path p to list_of_decorated_pps
    return list_of_decorated_pps;
  End

  Procedure: Append_data_transfers_to_pipeline_paths(decorated_pps,data_transfers)
  Input: The decorated_pps and data_transfers
  Output: The traces
  Begin
    for each decorated pipeline path p
       for all permutations of data transfers dt connected to the ports decorating p
          append all data transfers from dt to p
          add p to list_of_traces
    return list_of_traces;
  End

Algorithm: Phase-II-Bind-Operations-to-Traces
Input: the operation (opcode, unit, operands, format), and the traces in the architecture
Output: the RT for that operation
Begin Phase II
  temp_trace_list=Find_traces_for_unit_and_format(traces,unit,format);
  for each trace t in list temp_trace_list
    if(all_units_support_opcode(t,opcode) and
      operation_format_supported(t,format))
       Return_reservation_table(t);
End Phase II
```

Fig. 4. RTGEN algorithm. (a) Phase-I-generate-traces. (b) Phase-II-bind-operations-to-traces.

since the mapping of traces to specific operation is performed only in the second phase, this trace is not linked to any operation yet.

In the second phase, given an operation, we use the traces, operation format, and opcode-to-unit mapping to generate the corresponding RT. Each RT represents the architectural resources used in each pipeline stage by a particular operation. In this phase, we can use different strategies (with varying computation time and memory requirements) to generate the RTs, as explained in Section VI.

We use traces as an intermediate output of our algorithm because the number of traces in a typical processor is small compared to the number of fully qualified operations and, consequently, the number of RTs. Fig. 4 details the two phases of RTGEN.

Phase I: First, since EXPRESSION contains a hierarchical description of the pipeline, we flatten out the hierarchy into a set of distinct pipeline paths. For instance, one flattened pipeline path in the TI C6201 is PG, PS, PW, PR, DP, DC, M1_E1, and M1_E2 (Fig. 3). Also, since the data transfers are described as individual segments, we compose them into complete RF-to-FU and FU-to-RF transfers.[3] These two steps are necessary to transform the pipeline and data transfers description from the hierarchical EXPRESSION format to the format required by our algorithm. The rest of the RTGEN algorithm is independent of EXPRESSION and can be used with any processor ADL containing the features described in Section IV.

[3]RF stands for register files (e.g., RFA) and FU for functional units (e.g., M1_E1).

```
Stage 1: PG, Stage 2: PS, Stage 3: PW, Stage 4: PR, Stage 5: DP, Stage 6: DC
Stage 7: RFB, RFB_CROSSPATH, M1_E1_S1, RFA, RFA_M1_E1_S2_CONNECT,
                                                        M1_E1_S2, M1_E1
Stage 8: M1_E2, M1_E2_D, RFA_M1_E2_D_CONNECT, RFA
```

Fig. 5. Resources used in each stage by the trace highlighted in Fig. 3.

Next, procedure *Decorate_pipeline_paths()* annotates the units in each pipeline path with the ports corresponding to each data transfer involving that unit. For example, the M1_E1 unit from Fig. 3 is decorated with the ports M1_E1_S1 (for the SRC1 operand) and M1_E1_S2 (for the SRC2 operand), allowing transfer of data from RFA and RFB to M1_E1 (for conciseness, we did not represent these ports in the figure).

The ports annotating each pipeline path are chosen based on the operation formats. In order for a pipeline path to support an operation format, all the operands referenced in the operation format have to be implemented by a (unique) port decorating that pipeline path. For example, for the format *OPCODE FU SRC1 SRC2 DST* and for the pipeline path PG, PS, PW, PR, DP, DC, M1_E1 and M1_E2, SRC1 is covered by the port M1_E1_S1, SRC2 by M1_E1_S2, and DST by M1_E2_D. Thus, for each pipeline path, we try to satisfy each operation format by generating all possible decorations so that each operand in the format is covered by exactly one port. If for a pipeline path there are multiple formats supported, or multiple ways to decorate it, we duplicate the pipeline path, and use a different combination of ports to decorate each copy.

Finally, *Append_data_transfers_to_pipeline_paths* generates the traces by attaching a data transfer to each port decorating the pipeline paths. For a given port there may be multiple data transfers, which can be attached. For example, M1_E1_S1 can be used to transfer data between the RFA and M1_E1, or between the RFB and M1_E1, by following a different set of connections. For each port, we consider all the possible data transfers by duplicating the decorated pipeline paths and choosing a combination of data transfers to attach. The result of this step is called traces. One such trace in our example architecture is shown in Fig. 5. It contains the units PG, PS, PW, PR, DP, DC, M1_E1 and M1_E2 and the data transfers RFB to M1_E1, RFA to M1_E1, and M1_E2 to RFA. The unit M1_E1 reads one operand from RFB through the cross connection RFB_CROSSPATH and the port M1_E1_S1, and another one from RFA through the connection RFA_M1_E1_S1_CONNECT, and the port M1_E1_S1. The unit M1_E2 writes the result to RFA through the connection RFA_M1_E2_D_CONNECT, and the port M1_E2_D.

Phase II: In the second phase, we traverse the traces in order to find the trace corresponding to the input operation. We first narrow down the choice of traces to the ones corresponding to that operation's format and unit. For example, for the operation *MPYM1_E1 RFB0 RFA0 RFA0*, we choose only the traces that contain the M1_E1 unit and support the format *OPCODE FU SRC1 SRC2 DST*.

We then verify that all the units in the trace support the operation's opcode. In this way, we account for cases when different opcodes supported by one FU require different sets of resources. For example, the D1_E1 unit supports both LD and ADD operations, but LD additionally requires the MEM_CONTROLLER unit, whereas ADD does not. As the MEM_CONTROLLER does not support the ADD opcode, the traces containing the memory controller are excluded while determining RTs for the ADD operation.

Next, we choose the traces that satisfy the order in the operation's format. For example, for the operation *OPCODE FU SRC1 SRC2 DST*, SRC1 and SRC2 are read before DST is written.

At this point, the choice of traces corresponding to that operation has been narrowed down to one or more traces. A fully qualified operation corresponds to exactly one trace (which represents the RT for that operation) and is returned as the result of the RTGEN algorithm.

TABLE I
RT GENERATION

Arch.	EXPR. lines	Pipeline paths	DT paths	Traces	RTs	Time (s)	
						Traces	RTs
DLX_P	411	4	5	16	155	0.11	2.71
DLX_P_2RF	480	4	8	80	952	0.44	88.41
C62_1RF	1064	12	34	56	168	0.60	18.88
C62_2RF	1095	12	44	98	426	1.04	172.59

For example, given the fully qualified operation *MPY M1_E1 RFB0 RFA0 RFA0*, the trace returned as the RT by RTGEN is the one shown in Fig. 5.

During compilation, the operations are qualified incrementally. Starting from a generic (nonqualified) operation, the fields are bound one by one (depending on the phase ordering of that particular compiler) until the operation is fully qualified. In the case of partially qualified operations (e.g., the opcode and FU fields have been bound, but the argument fields have not been bound to the RFs yet), RTGEN computes a list of RTs corresponding to all the possible bindings of the not-yet-qualified fields of that operation (e.g., the RTs for the operations having the given opcode and FU, and all the possible RF choices for the source and destination operands). RTGEN can also provide an approximate RT, computed as the intersection (optimistic) or the union (conservative) of the list of possible RTs, thus providing conflict information even in the absence of complete information, at any level during the compilation process, making RTGEN independent of the phase ordering in the compiler.

The worst case complexity of Phase I of RTGEN is $O(x * z * \binom{y}{w})$, where x is the number of pipeline paths, y is the number of data transfers, z is the number of distinct operation formats for that processor, and w is the maximum number of operands in an operation. Typically, most architectures have a maximum number of operands from 3 to 4, and the number of formats is under ten. Moreover, very few of the choices in $\binom{y}{w}$ are considered. For example, while considering the choices of a data transfer for a particular operand, only those corresponding to the operand type (e.g., SRC1) and to the FU assigned to that operation are chosen. The worst case complexity of phase II of the algorithm is $O(m * n)$, where m is the number of traces in the architecture, and n is the number of fully qualified operations. Since this is the more time consuming part of the algorithm, we present in Section VI a discussion exploring different strategies to tradeoff computation-time against memory.

VI. EXPERIMENTS

We now present a set of experiments conducted on various processor descriptions and the generated RTs to drive pipelined scheduling of a set of multimedia and scientific benchmarks.

Table I presents the results of the RTGEN on the TI C62 processor and a multiissue version of the DLX processor. In the context of architectural DSE, we also present variants of each architecture to show how modifying features of the architecture (such as the register file architecture) impacts the number of traces and RTs.

The C62 processor is a VLIW DSP, allowing eight operations to be issued per cycle. The multiissue DLX architecture allows four operations per cycle and has a pipeline with up to 11 stages and multicycled units. The first column in Table I describes the architectures for which the RTs were generated automatically. We experimented with a single-register file (C62_1RF, DLX_1RF) version and a two-register file (C62_2RF, DLX_2RF) of the architectures. In C62_2RF (the actual C6201 architecture, also shown in Fig. 1), the two-register files are partitioned, with limited connectivity between FUs and RFs. The DLX_2RF contains two-register files that are connected to all the functional units.

TABLE II
RTGEN STRATEGIES FOR DLX AND TI C6201

Arch	Domain (# Apps.)	O-T-F	Precompute DB		Cached	
		Time[s]	Time[s]	DB size	Time[s]	DB size
DLX	Array (1)	2.4	2.7	155	0.3	15
	Matrix (2)	11.9			0.6	18
	MM (4)	50.0			1.4	23
	Numeric (8)	62.9			2.2	18
	Mixed (16)	153.0			4.9	23
C62	Array (1)	86.5	18.9	168	0.6	21
	Matrix (2)	387.0			1.3	28
	MM (4)	1735.1			3.2	30
	Numeric (8)	2440.4			6.2	28
	Mixed (16)	5378.3			12.2	30

The second column shows the number of EXPRESSION lines specifying the complete architecture (including structure and ISA). The third column shows the number of pipeline paths, while the fourth column shows the number of data-transfer paths in the processors. The fifth and sixth columns show the number of traces and RTs generated, while the last two columns present the computation time needed to automatically generate the traces and RTs.

It is important to note that, in order to compile accurately for an architecture, all resource constraints (in our case, RTs) have to be either specified or generated. Especially in the case of orthogonal architectures, as each RT corresponds to a fully qualified operation, the number of RTs may be very large (e.g., 952 for DLX_2RF). Manually specifying RTs on a per-operation basis is very tedious and may lead to increased errors in specification. Furthermore, simple changes during architectural DSE may affect many RTs, requiring the re-specification of some or all RTs; in our approach, we only need to re-specify the architecture as the modified RTs are generated automatically. The importance of RTGEN is that it can handle real-life processors, including VLIW and Superscalar architectures, avoiding manual specification and updating of this large number of RTs.

In C62_1RF and DLX_1RF, the number of RTs is 168 and 155. On the other hand, for the two register file versions, it increases to 426 and 952. This is due to the fact that in the 2RF versions, the operations may read their operands from two possible locations, leading to a larger number of fully qualified operations. The significant difference in RTs between DLX_2RF and C62_2RF is due to the different RF architecture. For the DLX, the operands of any operation can belong to any of the 2RFs, while for the C62_2RF, the restricted connectivity between the FUs and RFs precludes many operand combinations.

To deal with the large number of RTs, in the following, we explore different strategies trading off computation time and memory requirement. Recall that we generate the RTs in two phases: first, we extract a set of traces, modeling the execution patterns of the operations; second, we bind these execution traces to individual operations, in order to generate RTs on a per-operation basis. This separation of concerns allows us to make some interesting tradeoffs between time and memory requirements during RTGEN.

Phase I of RTGEN—extraction of traces—can be performed rather quickly (the seventh column in Table I). As can be seen, it is in the order of seconds, even for a relatively complex architecture like the C62. Phase II of RTGEN—binding of RTs to operations—is the more time-consuming step. We present three strategies, which have varying time and memory requirements. The first, called *on-the-fly (O-T-F)*, binds RTs as and when required by tools. The second, called *precompute database*, binds RTs for all operations before hand and stores them in a database. The third, called *cached*, is a modified O-T-F approach with RTs generated on demand, but stored in a database for future access.

In Table II, we present the tradeoffs in terms of computation time and memory requirement for the generation of RTs using the three strategies. We ran our algorithm on five sets of benchmarks containing 1, 2, 4, 8, and 16 applications from a suite of multimedia and scientific applications, containing filters (e.g., wavelet), image processing (e.g., Laplace edge enhancement), and numeric code (e.g., linear recurrence equation solvers, successive over-relaxation, red–black Gauss–Seidel relaxation). For details, please refer to [12].

In Table II, column 2 shows the application domain and the number of applications in each benchmark set. Column 3 shows the total time required to generate the RTs *on-the-fly* for a parallelizing compiler. Columns 4 and 5 show the total time (this includes the time required to precompute the RTs, but not the time to retrieve them from the database) and database size (number of RTs) needed in the *precompute DB* approach, while columns 6 and 7 show the total time and maximum database size for the *cached* approach.

As expected, the time required to generate RTs using the naive O-T-F approach is very large. However, it requires minimal memory, as it does not store any RTs. The database approach works best when compiling many applications since the one-time DB computation is better amortized. However, the memory penalty is large when compared to the other approaches. The cached version results in significant time improvement (as compared to O-T-F) and memory reduction (as compared to database). For example, for DLX, the cached approach performs well for small sets of benchmarks (1, 2, 4), while the database approach performs better for large sets (16).

Possible improvements to the overall approach may address optimizing the RT representation for compactness and speeding-up the conflict detection process. While the RTs generated are accurate, and lead to an efficient schedule and good pipeline resource utilization, they are not optimized from the point of view of compilation time. These issues are orthogonal to RTGEN and can be coupled with RTGEN. Reference [6] present a hierarchical description of RTs to optimize the description size. Tables of conflicts, containing the illegal combinations of operations, as well as state diagrams [1] capturing the state of the current schedule can be generated from RTs. They speed up the conflict detection by replacing the comparison of RTs with a table lookup, or a transition in the FSA. However, RTs are required for generation of both the lookup tables and FSAs.

Moreover, by removing the redundancy from the RTs, such as resources that do not generate scheduling conflicts, or resources that generate redundant conflicts (conflicts that have been already exposed by previous resources), it is possible to reduce the size of the RTs and increase the speed of the compilation. It is possible to arrive to a set of "minimal" RTs, which contain the minimal amount of resources that are enough to allow accurate scheduling.

The two-tiered approach to automatic RTGEN allows the system designer to experiment with these various approaches depending on the objectives during DSE. The combined benefits of automatic RTGEN and the flexible approach to generation/usage of these RTs allows the system designer to significantly reduce the time spent in RT (re-specification) specification during architectural DSE.

VII. Summary

RTs are needed to detect conflicts between operations (e.g., two operations trying to use the same unit at the same time). RTs have been used for a long time to drive scheduling in the compiler and to generate state diagrams driving dynamic scheduling in the hardware controller. In this paper, we have presented RTGEN, an algorithm to automatically generate RTs from an architecture description of the processor.

Our approach bridges the gap between the structural representation of processors, typically used by processor designers, and the higher level information needed by the compilers. Traditionally, detailed RTs were specified by hand. Due to the increasing complexity of today's processors, containing hundreds of operations, extensive parallelism (e.g., TIC6X), and deep pipelines, specifying RTs by hand is a very laborious and error-prone task. Moreover, during architectural exploration, in order to keep the compiler up-to-date with the processor, the designer needs to reflect the changes to the architecture in the RT specification. This is a very tedious task. By automatically generating RTs, we avoid the need for explicit specification, and we support fast architectural exploration, by automatically reflecting the changes to the architecture in the compiler.

We have presented a set of experiments on the TI C6201 VLIW DSP, as well as on the DLX architecture. Our prototype tool starts from an EXPRESSION description, and generates RTs for the EXPRESS compiler. We have presented three RTGEN strategies with varying time and memory requirements. The experiments show the results on a set of multimedia and scientific kernels. Future work will investigate other RTGEN strategies and will also apply these techniques to a wider class of architectures.

Acknowledgment

The authors would like to acknowledge and thank N. Savoiu, R. Cornea, P. Mishra, S. Srinivasan, P. Biswas, and A. Shrivastava for their contributions to the EXPRESS/EXPRESSION project.

References

[1] V. Bala and N. Rubin, "Efficient instruction scheduling using finite state automata," *Int. J. Parallel Processing*, vol. 25, no. 2, pp. 53–82, 1997.
[2] E. S. Davidson, "The design and control of pipelined function generators," in *IEEE SNC Conf.*, 1971, pp. 19–21.
[3] A. E. Eichenberger and E. S. Davidson, "A reduced multipipeline description that preserves scheduling constraints," in *Programming Languages, Design and Implementation*, May 1996, pp. 12–20.
[4] E. S. Davidson *et al.*, "Effective control for pipelined processors," in *IEEE COMPCON*, San Francisco, CA, 1975, pp. 181–184.
[5] G. Hadjiyiannis *et al.*, "ISDL: An instruction set description language for retargetability," in *Proc. Design Automation Conf.*, 1997, pp. 299–302.
[6] J. C. Gyllenhaal *et al.*, "Optimization of machine descriptions for efficient use," *Int. J. Parallel Processing*, vol. 26, no. 4, pp. 417–447, 1998.
[7] P. G. Lowney *et al.*, "The multiflow trace scheduling compiler," *J. Supercomput.*, vol. 7, pp. 51–142, 1993.
[8] V. Zivojnovic *et al.*, "LISA—Machine description language and generic machine model for HW/SW co-design," in *Proc. IEEE VLSI Signal Processing Workshop*, 1996, pp. 127–136.
[9] M. Freericks, "The nML machine description formalism," Comput. Sci. Dept., Tech. Univ. Berlin, Berlin, Germany, Tech. Rep. TR SM-IMP/DIST/08, 1993.
[10] P. Grun, N. Dutt, and A. Nicolau, "Memory aware compilation through accurate timing extraction," in *Proc. Design Automation Conf.*, Los Angeles, CA, 2000, pp. 316–321.
[11] ——, "MIST: An algorithm for memory miss traffic management," in *IEEE/ACM Int. Computer-Aided Design Conf.*, San Jose, CA, 2000, pp. 431–437.
[12] P. Grun, A. Halambi, N. Dutt, and A. Nicolau, "Automatic generation of reservation tables from an architecture description," Univ. California at Irvine, Irvine, CA, Tech. Rep., 1999.
[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 1996.
[14] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "V-SAT: A visual specification and analysis tool for system-on-chip exploration," in *Proc. EUROMICRO Conf.*, Oct. 1999, pp. 196–203.
[15] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
[16] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions," *Des. Autom. Embedded Systems*, vol. 3, no. 1, pp. 75–108, 1998.
[17] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann, 1997.
[18] A. Nicolau and S. Novack, "Trailblazing: A hierarchical approach to percolation scheduling," presented at the ICPP, St. Charles, IL, 1993.

[19] S. Novack, A. Nicolau, and N. Dutt, "A unified code generation approach using mutation scheduling," in *Code Generation for Embedded Processors*. Boston, MA: Kluwer, 1995.
[20] T. A. Proebsting and C. W. Fraser, "Detecting pipeline hazards quickly," *PPL*, Jan. 1994.
[21] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Microarchitecture*, Nov. 1994, pp. 63–74.
[22] C. Siska, "A processor description language supporting retargetable multi-pipeline DSP program development tools," in *Proc. Int. System Synthesis Symp.*, Dec. 1998, pp. 31–36.
[23] *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Texas Instruments Incorporated, Dallas, TX, 1998.
[24] (1997) *The MDES User Manual, Trimaran Release*. [Online]. Available: http://www.trimaran.org.

Design of a Cycle-Efficient 64-b/32-b Integer Divisor Using a Table-Sharing Algorithm

Chua-Chin Wang, Po-Ming Lee, Jun-Jie Wang, and Chenn-Jung Huang

***Abstract*—In new generations of microprocessors, the superscalar architecture is widely adopted to increase the number of instructions executed in one cycle. The division instruction among all of the instructions needs more cycles than the rest, e.g., addition and multiplication. It then makes division instruction an important cycles-per-instruction figure for modern microprocessors. In this paper, a radix-16/8/4/2 divisor is proposed, which uses a variety of techniques, including operand scaling, table partitioning, and, particularly, table sharing, to increase performance without the cost of increasing complexity. A physical chip using the proposed method is implemented by 0.35-μm single poly four metal (1P4M) CMOS technology. The testing measurement shows that the chip can execute signed 64-b/32-b integer division between 3–13 cycles with a 80-MHz operating clock.**

***Index Terms*—Integer division, mixed radixes, on-the-fly conversion, operand scaling, table folding, table sharing.**

I. Introduction

Integer division is a critical operation in CPU design since the number of clock cycles to complete an integer division is probably very long and unpredictable. The role of division is becoming more and more critical owing to the requirement of signed computer arithmetics, the modulus computation, the calculation of encryption keys, and so on. Division algorithms can be roughly classified into two categories, namely, digit-recurrence methods [1] and functional iteration techniques [1], while the former is commonly used. Regarding the digit-recurrence method, traditionally there are two types of division schemes, i.e., restoring and nonrestoring schemes. However, they both require multiple operation steps to derive a quotient bit. Not only is the efficiency drastically poor, but also a long adder/subtracter is needed to execute the remainder bit adjustment.

In this paper, we employ a modified high-radix, i.e., radix-16/8/4/2, digit-recurrence division method and on-the-fly conversion method to reduce the required cycles for the 64-b/32-b signed integer division, while keeping the hardware complexity in control.

Manuscript received March 30, 2001; revised April 30, 2002. This work was supported by the Academic Foundation of the Taiwan Ericsson Company Ltd. This work was supported in part by the National Science Council under Grant NSC 89-2218-E-110-014 and Grant NSC 89-2218-E-110-015.

C.-C. Wang and P.-M. Lee are with the Department of Electrical Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan 80424, R.O.C. (e-mail: ccwang@ee.nsysu.edu.tw).
J.-J. Wang is with VIA Technologies Inc., Taipei, Taiwan, 231 R.O.C.
C.-J. Huang is with the Department of Computer Science and Information Education, National Taitung Teachers College, Taitung, Taiwan 95004, R.O.C. (e-mail: cjh@cc.ntttc.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2003.816143

II. High-Radix 64-b/32-b Signed Integer Divisor

A. Digit-Recurrence Theory

Assume x, d, q, and rem to be the dividend, divider, quotient, and remainder in the division operation. We also denote the radix of the division as being r. The division is then defined as $x = q \cdot d + \mathrm{rem}$. In the digit-recurrence division algorithm [2], 1–b bits of quotient digit can be obtained every iteration in a radix-2^b digit-recurrence division. In other words, b bits of quotient can be obtained every iteration. In [3], the digit-recurrence algorithm is defined as

$$w[j+1] = r \cdot w[j] - d \cdot q_{j+1} \tag{1}$$

where $w[j+1]$ is the residual of the $(j+1)$th iteration, r is the radix, and q_{j+1} is the quotient digit generated in the $(j+1)$th iteration. In a radix-r, $r = 2^b$, division, the quotient digit set is defined as $q_j \in D_a = \{-a, \ldots, -1, 0, 1, \ldots, a\}$. Since $\|D_a\| > r$, it uses more than r numbers to present the quotient digits, which make this quotient representation form to be a redundant form. Besides, the restriction of a is $a \geq \lceil r/2 \rceil$. In (1), the quotient digits are generated in every iteration. Hence, we can define the quotient-digit selection function as $q_{j+1} = \mathrm{SEL}(w[j], d)$, where the $\mathrm{SEL}()$ function can be simplified as a table lookup function.

Although the digit-recurrence algorithm has been well written in [2], there are many unsolved difficulties when it comes to hardwaredly realizing such a divisor, including the following.

1) A long adder is needed at the adjustment of the remainder.
2) Extra adjustment actions are required when the last cycle of the division contains nonmultiple digits of the radix. (For instance, the radix is 16, but there is only 1 b left in the dividend to be processed.)
3) The adjustment of the remainder is missing when the signed division is executed.
4) A data flow control unit is required, which provides correct timing control such that the results of the division can be correctly placed on the output ports.
5) The size of the quotient selection table will grow exponentially with the radix. Besides, it is likely that one radix needs one table. These two factors lead to a huge chip area consumption if the divisor is implemented on silicon.

In short, the above problems will occur during the realization of a long signed divisor. If these problems are not resolved efficiently, the hardware divisor will be large and slow.

B. Mixed Radix-16/8/4/2 64-b/32-b Integer Divisor

In [4], a mixed radix-8/4/2 integer divisor was proposed, of which performance is better than that of a normal radix-4/2 integer divisor [5]. However, it paid the price of increasing the complexity of hardware, and then nearly doubled the total area of the divisor owing to the sizes of tables. In this study, despite that the radix will be raised up to 16 to retire more bits of the quotient per cycle, the complexity of the hardware will be retained to a similar degree by using several methods, including operand prescaling, table partitioning, [6], and table folding.