RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions*

Peter Grun Ashok Halambi Nikil Dutt Alex Nicolau

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA
http://www.cecs.uci.edu/~aces

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 turn-around time in Design Space Exploration. We demonstrate the utility of our approach on a set of experiments using the TI C6201 VLIW DSP and DLX processor architectures, and a suite of multimedia and scientific applications.

1 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 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 the 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. Since RTs can be specified at different levels of detail, they can also be used in an architectural Design Space Exploration (DSE) environment involving trade-offs between accuracy and speed of the software tools for embedded Systems-on-Chip.

Most current retargetable tools that follow the RT 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 **RTGEN**, an algorithm that 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. Since every operation proceeds through a pipeline path and accesses storage units through some data-transfer paths, the key idea behind the RT generation approach is that it is possible to trace the execution of the operation through the architecture's pipeline and data-transfer segments and thus generate accurate RTs.

In Section 2, we describe related work on ADL-driven pipeline (and constraint) specification for tools and compare them with our approach. In Section 3, we motivate the need for automatic RT generation using the TI C6201 VLIW DSP. Section 4 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 5 presents the algorithm for automatic RT generation. Section 6

*This work was partially supported by grants from NSF (MIP-9708067) and ONR (N00014-93-1-1348).

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

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 7 concludes this paper.

2 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. [14] surveys pipelining techniques and provides most of the terminology and concepts in use today. [12] contains a good description of the various aspects of pipelining (including tackling hazards and compilation techniques).

The advent of System-on-Chip (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, Reservation Tables (RTs) are used to detect conflicts for scheduling [16]. 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 [22]. 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 Design Space Exploration (DSE) structural changes to the architecture may propagate through the description, requiring the user to manually reflect the changes in the RT section too.

State diagrams or Finite State Automatons (FSAs) are used to represent the set of all legal instruction schedules for a processor. The FSAs are derived from RTs. [1], [18] present compiler techniques that use FSAs. For determining operation conflicts, the RT approach suffers from the drawback of increased 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[14]. A disadvantage of the FSA approach is that it is not amenable to certain advanced scheduling techniques (such as iterative modulo scheduling[19] and mutation scheduling[17]).

The LISA [8] and RADL [20] 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 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. [15] presents a technique for automatic extraction of the instruction set, 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[13], as well as the RTs required by our optimizing compiler EXPRESS.

3 Motivating Example

We use Texas Instruments' C6201 VLIW DSP to intuitively explain the RT generation algorithm and illustrate the complexity of the problem. The C62 is a state-of-the-art fixed-point digital signal processor (DSP) with a high-performance, VLIW architecture, whose block-level diagram is shown in Figure 1. The bold blocks represent pipeline and functional units, while the dotted blocks represent storage components. The interesting features of this architecture include instruction-level parallelism (ILP) with up to 8 operations being issued in one cycle, a complex, fragmented pipeline and a complex storage subsystem with 2 register files (RFA, RFB) with multiple read/write ports and a main memory with 4 banks.

For ease of illustration, we have omitted showing the main controller, pipeline latches, ports and some connections and storages. However, the actual RT generation algorithm 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 [21].

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.

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

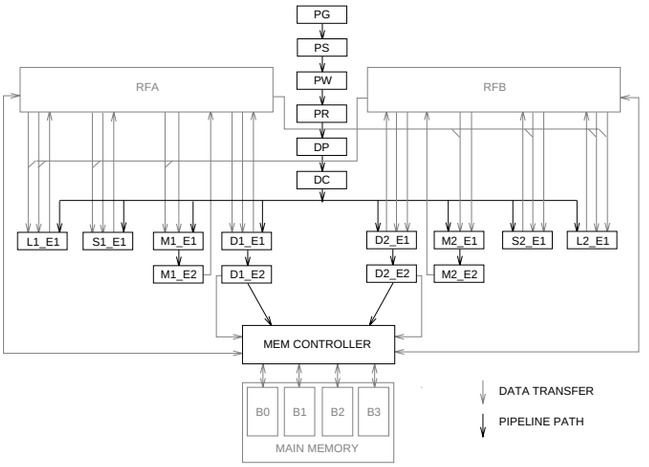

Figure 1. Block diagram of the TI C6201 VLIW DSP. PG, PS, PW, PR perform instruction fetch; DP, DC perform decode and dispatch; L, S, M, D are functional units.

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 RT generation 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).

4 ADL Information Required for RT generation

We now describe the essential features required in an Architecture Description Language (ADL) to support automatic generation of RTs. While we use our ADL EXPRESSSION [11] as a vehicle for demonstrating the automatic generation of RTs from a machine description, it is important to note that the RT generation 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 RT generation is integrated specification of both structure and behavior (i.e., instruction-set) 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 the *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 which 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 instruction-set. Each operation in the instruction-set 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).

The information captured in the ADL allows us to generate resource conflicts for multi-issue 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 the next section.

5 RTGEN: An Algorithm for Automatic Generation of Reservation Tables

Figure 2 presents the flow of **RTGEN**, the Reservation Tables (RT) generation algorithm. 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 instruction set 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 Figure 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 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 6.

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. Figure 4 details the two phases of RTGEN, the RT Generation algorithm.

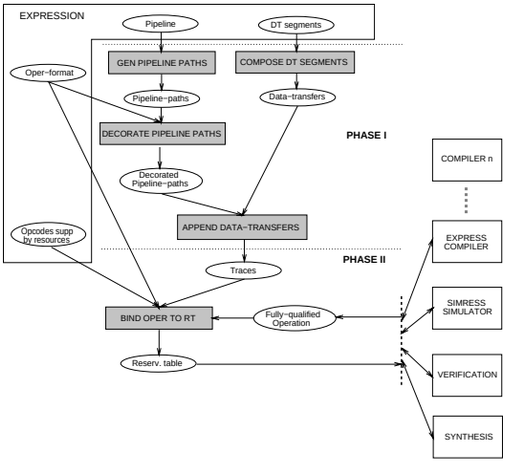

Figure 2. Overview of the RTGEN algorithm

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, M1_E2* (Figure 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 architecture description language containing the features described in Section 4.

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 Figure 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 port decorating that pipeline path. E.g., 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.

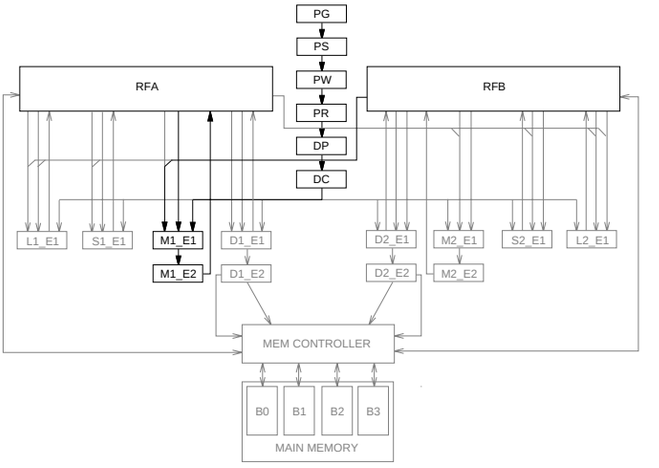

Figure 3. A trace in the TI C6201 architecture

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. E.g., 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 Figure 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. E.g., for the operation *MPY M1_E1 RFB0 RFA0 RFA0* we choose only the traces which contain the M1_E1 unit, and support the format *OPCODE FU SRC1 SRC2 DST*.

Then, we verify that all the units in the trace support the operation's opcode. In this way we account for cases

[3] RF stands for Register Files (e.g., RFA), and FU for Functional Units (e.g., M1_E1).

```
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
```

Figure 4. RTGEN Reservation Tables Generation Algorithm: (a) Phase-I-Generate-Traces, and (b) Phase-II-Bind-Operations-to-Traces

when different opcodes supported by one FU require different sets of resources. E.g., 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. E.g., 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. 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 Figure 5.

During compilation, the operations are qualified incrementally. Starting from a generic (non-qualified) 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 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.

```
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
```

Figure 5. The resources used in each stage by the trace highlighted in Figure 3

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 the number of data transfers, z is the number of distinct operation formats for that processor, and w the maximum number of operands in an operation. Typically, most architectures have a maximum number of operands between 3 and 4, and the number of formats is under 10. Moreover, very few of the choices in $\binom{y}{w}$ are considered. E.g., 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 6 a discussion exploring different strategies to trade-off computation-time against memory.

6 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.

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

Table 1. RT Generation

Table 1 presents the results of the RT generation algorithm on the TI C62 processor and a multi-issue version of the DLX processor. In the context of architectural design space exploration, 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 8 operations to be issued per cycle. The multi-issue DLX architecture allows 4 operations per cycle and has a pipeline with up to 11 stages and multi-cycled units. The first column in Table 1 describes the architectures for which the RTs were generated automatically. We experimented with a single register file (C62_1RF, DLX_1RF) version and a 2 register file (C62_2RF, DLX_2RF) of the architectures. In C62_2RF (the actual C6201 architecture, also shown in Figure 1) the 2 register files are partitioned, with limited connectivity between FUs and RFs. The DLX_2RF contains 2 register files which are connected to all the functional units.

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 design space exploration 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 2 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 2 RFs, 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 trade-offs between time and memory requirements during RT generation.

Phase I of RTGEN – extraction of traces – can be performed rather quickly (the seventh column in Table 1). 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.

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

Table 2. RT generation strategies for DLX and TI C6201

In Table 2 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 5 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 [10].

In Table 2, 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). E.g., 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. However, these issues are orthogonal to RT generation, and can be coupled with

RTGEN. [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 look-up, or a transition in the FSA.

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

7 Summary

Reservation Tables (RT) 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 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 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 presented 3 RT generation 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 RT generation strategies and will also apply these techniques to a wider class of architectures.

8 Acknowledgements

We would like to acknowledge and thank Asheesh Khare, Nick Savoiu, and Vijay Ganesh for their contributions to this work.

References

[1] V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. *IJPP*, 25(2):53–82, 1997.

[2] E. S. Davidson. The design and control of pipelined function generators. In *IEEE Conf. SNC*, pages 19–21, Mexico, 1971.

[3] A. E. Eichenberger and E. S. Davidson. A reduced multipipeline description that preserves scheduling constraints. In *PLDI*, pages 12–20, May 1996.

[4] E. S. Davidson et al. Effective control for pipelined processors. In *IEEE COMPCON*, pages 181–184, San Fransisco, 1975.

[5] G. Hadjiyiannis et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.

[6] J. C. Gyllenhaal et al. Optimization of machine descriptions for efficient use. *IJPP*, 26(4):417–447, 1998.

[7] P. G. Lowney et al. The multiflow trace scheduling compiler. *J. Supercomputing*, 7:51–142, 1993.

[8] V. Zivojnovic et al. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[9] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[10] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. Automatic generation of reservation tables from an architecture description. Technical report, University of California, Irvine, 1999.

[11] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, March 1999.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 1996.

[13] 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*, October 1999.

[14] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, New York, 1981.

[15] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[17] S. Novack, A. Nicolau, and N. Dutt. A unified code generation approach using mutation scheduling. In *Code Gen. for Embedded Processors*. Kluwer, 1995.

[18] T. A. Proebsting and C.W. Fraser. Detecting pipeline hazards quickly. In *PPL*, January 1994.

[19] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. 27th Microarchit.*, November 1994.

[20] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, December 1998.

[21] TEXAS INSTRUMENTS. *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, 1998.

[22] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.