AUTOMATIC DATA PATH GENERATION FROM C CODE FOR CUSTOM PROCESSORS

Jelena Trajkovic and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine
jelenat@cecs.uci.edu, gajski@cecs.uci.edu

Abstract The stringent performance constraints and short time to market of modern digital systems require automatic methods for design of high performance application-specific architectures. This paper presents a novel algorithm for automatic generation of custom pipelined data path for a given application from its C code. The data path optimization targets both resource utilization and performance. The input to this architecture generator includes application C code, operation execution frequencies obtained by the profile run and a component library consisting of functional units, busses, multiplexers etc. The output is data path specified as a net-list of resource instances and their connections. The algorithm starts with an architecture that supports maximum parallelism for implementation of the input C code and iteratively refines it until an efficient resource utilization is obtained while maintaining the performance constraint. This paper also presents an algorithm to choose the priority of application basic blocks for optimization. Our experimental results show that automatically generated data paths satisfy given performance criteria and can be obtained in a matter of minutes leading to significant productivity gains.

Keywords: Architecture, Data Path, Design, Synthesis, C-to-RTL, Pipeline, Performance, Utilization

1. Introduction

Performance requirements for modern applications have fueled a need for specialized processors for different application domains. The reference C code typically serves as a starting point for most designs. To meet design deadlines, automatic generation of design from reference C code is needed. For most modern applications such C references are typically in the order of thousands of lines of code which is beyond the capacity of existing C-to-RTL tools. Manual hardware design is expensive and error prone process. Even though the designs are tuned to satisfy stringent performance, area and power constraints,

reuse and feature extension are very difficult. On the other hand, the general purpose embedded processors are often too slow and power hungry, but offer programmability. Therefore, in this paper, we propose the automatic generation of the data path architecture based on the profile of the application and the system performance and utilization constraints. We follow a design technique for custom processors that separates the allocation of architectural resources and their connections from the scheduling of control words that drive that data path. Fig. 1 shows proposed design approach. The application code is first scheduled in an As Late As Possible (ALAP) fashion. After an application's requirements have been derived from ALAP-like schedule, they are used for allocation of the data path components. The resulting architecture is evaluated and refined using the Architecture Wizard. The Architecture Wizard iterates through the possible configurations until the given performance constraint and component utilization are satisfied.

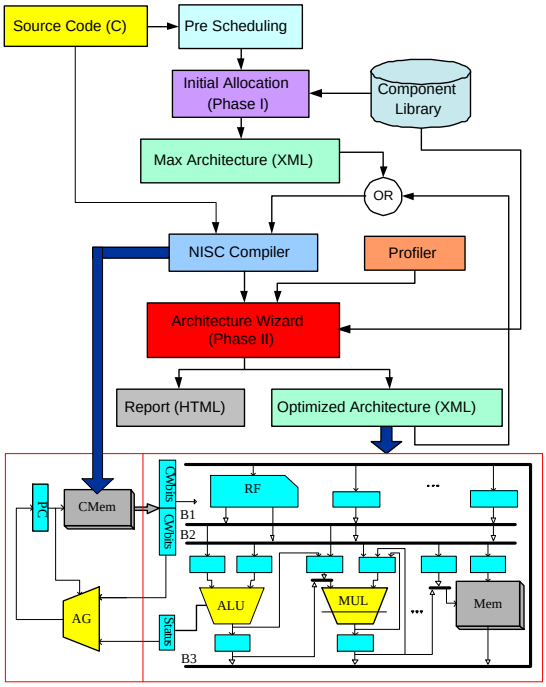

Figure 1. Custom Processor Design Technique.

2. Related Work

In order to accomplish performance and power goals, ASIP and IS extension use configurable and extensible processors. One such processor is Xtensa [Tensilica: Xtensa LX, 2005], that allows the designer to configure features

like memories, external buses, protocols and commonly used peripherals [Automated Configurable Processor Design Flow, 2005; Diamond Standard Processor Core Family Architecture, 2006]. Xtensa also allows the designer to specify a set of instruction set extensions, hardware for which is incorporated within the processor [Goodwin and Petkov, 2003]. The extensions are formed from the existing instructions in style of VLIW, vector, fused operations or combination of those. Therefore, the customizations are possible only within bound of those combinations of existing instructions. This solution also requires the decoder modifications in order to incorporate new instructions. For example, having VLIW-style (parallel) instructions require multiple parallel decoders [Goodwin and Petkov, 2003], which not only increase hardware cost (that may affect the cycle time), but also limits the possible number of instructions that may be executed in parallel. However, in our approach, the decoding stage has been removed. Therefore, there is no increase in hardware complexity and no limitations on number and type of operations to be executed in parallel. In case where the code size exceeds the size of on-chip memory, instruction caches and compression techniques may be employed, both of them have been in scope of our current research.

The IS extensions, in case of Stretch processor [Stretch. Inc.: S5000 Software-Configurable Processors, 2005], are implemented using configurable Xtensa processor and Instruction Set Extension Fabric (ISEF). The designer is responsible for, using available tools, identifying the critical portion of the code ('hot spot') and re-writing the code so the 'hot spot' is isolated into the custom instruction. The custom instruction is then implemented in ISEF. Thus, the application code needs to be modified which requires expertise and potentially more functional testing. The designer is expected to explicitly allocate the extension registers. The ISEF and the main processor have the same clock cycle, and only one custom instruction/function may be generated. In contrary, our approach allows, but does not require C code modifications and does not require the designer to manipulate the underlying hardware directly.

Many C-to-RTL algorithms, such as [B. Landwehr et al., 1994; Paulin and Knight, 1989] create data path while performing scheduling and binding. [B. Landwehr et al., 1994] uses ILP formulation with emphasis on efficient use of library components, which makes it applicable to fairly small input code. [Paulin and Knight, 1989] tries to balance distribution of operations over the allowed time in order to minimize resource requirement hence the algorithm make decisions considering only local application requirements. [Devadas and Newton, 1989] takes into account global application requirements to perform allocation and scheduling simultaneously using simulated annealing. In contrast with the previous approaches, we separate data path creation from the scheduling and/or binding i.e. controller creation. This separation allows us to potentially reuse created data path by reprogramming, have controllability

over the design process and use pre-layout information for data path architecture creation.

[Gutberlet et al., 1992; Tseng and Seiwiorek, 1986; Tsai and Hsu, 1992; Brewer and Gajski, 1990; Marwedel, 1993] separate allocation from binding and scheduling. [Gutberlet et al., 1992] uses 'hill climbing' algorithm to optimize number and type of functional unit allocated, while [Tseng and Seiwiorek, 1986] applies clique partitioning in order to minimize storage elements, units and interconnect. [Tsai and Hsu, 1992] use the schedule to determine the minimum required number of functional units, buses, register files and ROMs. Than, the interconnect of the resulting data path is optimized by exploring different binding options for data types, variables and operations. In [Brewer and Gajski, 1990] the expert system breaks down the global goals into local constraints (resource, control units, clock period) while iteratively moves toward satisfying the designer's specification. It creates and evaluates several intermediate designs using the schedule and estimated timing. However, all of before-mentioned C-to-RTL techniques use FSM-style controller. Creation and synthesis of such state machine that corresponds to thousands of lines of C code, to the best of our knowledge, is not practically possible. In contrast to this, having programmable controller, allows us to apply our technique to (for practical purposes) any size of C code, as it will be shown in Section 6.

Similarly to our approach, [Marwedel, 1993] does not have limitations on the input size, since it uses horizontally microcoded control unit. On the other hand, it requires specification in language other than C and it produces only non-pipelined designs, none of which is the restriction of the proposed technique.

3. Proposed Approach

We propose a custom processor design technique for the No-Instruction-Set Computer (NISC) [Gajski, 2003]. NISC completely removes the decoding stage and stores the control words in the program memory. The NISC compiler ([Reshadi and Gajski, 2005], [Reshadi et al., 2005]) compiles the application directly onto a given data path, creating a set of control signals (called control word) that drives the components at runtime. By not having the instruction set, the data path can be easily changed, parameterized and reconfigured. Hence, the NISC concept allows separation of scheduling and allocation.

The tool flow that implements the proposed methodology is described in Fig. 1. It consists of 2 phases: the first one is performed by *Initial Allocation* and the second is implemented by the *Architecture Wizard* (*AW*) tool. In the Initial Allocation phase (Section 4), we use the schedule information to analyze component and connection requirements, and the available parallelism of a given application. The component and connection requirements are then taken

into account while choosing the instances of the available components from *Component Library* (*CL*) that will implement the data path. Resulting architecture is called Max Architecture. The Max Architecture and the application source code are used by the NISC compiler to produce the *new schedule*. The new schedule, results of the profile run and the component library are fed to the Architecture Wizard that performs estimation and refinement (Section 5). The Architecture Wizard evaluates component utilization, and uses it together with given performance and utilization constraints, to refine the existing data path architecture. The Architecture Wizard also estimates the potential performance overhead and utilization for the 'refined' architecture and automatically updates the new architecture if constraints are not satisfied. It outputs the netlist of the optimized architecture and the report in the human readable format.

4. Initial Allocation

We start by defining maximal requirements of a given application from the application's ALAP schedule (produced by Pre-Scheduler). We choose ALAP because it gives good notion of the operations that may be executed in parallel. In addition to application's schedule, component library is another input of the Initial Allocation. The Allocator traverses the given schedule, collecting the statistics for each operation. For each operation (addition, comparison, multiplication etc.) maximum and average number of its occurrences in each cycle is computed. The tool also records the number of potential data transfers for both source and destination operands.

The Component Library consists of resources, where each one is indexed by their unique name and identifier. The record for each component also contains its type, number of input or output ports and name of each port. In case of a functional unit, a hash table is used to link the functional unit type with the list of all operations it may perform.

We derived heuristics that measure how well the available storage components match the given requirements. The heuristics use required number of source and destination operands and number of output and input ports for the storage elements available in the CL [Trajkovic et al., 2006].

While allocating functional units, we choose the type of unit that alongside the given operation, performs the largest number of operations. Thus we prevent allocation of too many units and allow the Architecture Wizard to collect statistics of operations used and potentially replace the unit with the simpler one. Once the type is decided, we allocate maximum number that is computed by the Initial Allocation tool. For example, if application requires 3 additions and 4 subtractions, and the ALU is chosen, the tool will allocate 4 instances of ALU. For practical purposes we do not allow the number of allocated units of each type to exceed the number of source buses.

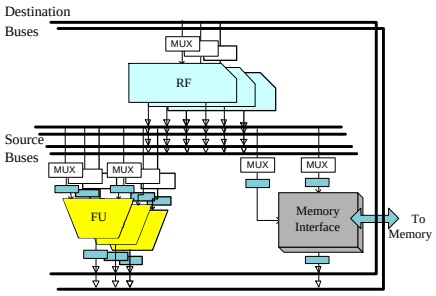

Figure 2. Components and connections.

To ensure that the interconnect is not a bottleneck in the Max Configuration, we perform a greedy allocation of connection resources (Fig. 2). This means that output ports of all register files are connected to all source buses. Similarly, input ports of all register files are connected to all destination busses. The same connection scheme applies to the functional units and the memory interface.

5. Estimation and Refinement

The Architecture Wizard (AW) attempts to reduce the number of used resources to create the final design that meets given performance and utilization goals. The source code is first compiled using the Max Architecture. The resulting schedule which also has the binding information together with the execution frequencies of each basic block from the profile run is used by the AW. The following algorithm shows main steps that the AW implements.

Extract Critical Path
Create Histogram
for all Basic Block and Component Type
Label
Flatten Histogram
Estimate Overhead and Utilization
if (Overhead >= Desired Overhead and Utilization <= Desired Utilization)
Update Number of Instances of Component
Goto **Label**
Allocate Components and Create Net-list

We start by selecting the basic blocks that we want to optimize. Next, we create the histogram for each functional unit type or for each group of input or output ports of the storage unit. It is necessary to consider the utilization of all components (functional units, storage components and buses) of the same type in order to apply 'Spill' (flattening) algorithm described in the Section 5.3. For the selected blocks, we estimate the number of instances of the chosen component that will keep the execution within a given boundaries and utilization. This is repeated until both the performance and utilization constraints are met. Once the optimal number of components is decided, the output net-list is created. The following sections describe the main steps of the AW in more details.

5.1 Selection of Basic Blocks for Optimization

The goal of this phase of the Architecture Wizard is to select the basic blocks in the source code that contribute the most to the execution time and have the largest potential for optimization. The question is how to decide which basic blocks are the most promising. Our selection criteria is based on the relative size and the relative execution frequencies of the basic blocks in the application. It is likely that very large basic blocks have high potential for optimization, since they have several operations that may potentially be performed in parallel. On the other hand, even minor optimization of basic blocks that have high frequency will yield high overall performance gains. Finally, we have a class of basic blocks that have average length and average frequency, so an average reduction in their length will yield overall performance improvement that is comparable to the improvement from previous two types of optimization.

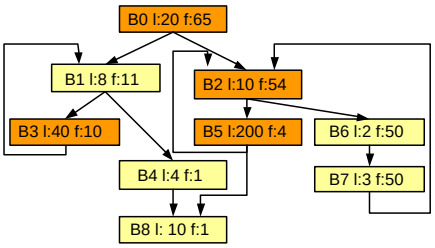

Figure 3. Selection of basic blocks for optimization.

For the Max Architecture, we use a profiler to record the execution frequency of each basic block and we use the schedule (histogram) to find the basic block's length. Following our optimization policy, we keep 3 lists of pointers to the basic blocks. The first list is sorted by frequency, the second by length and the third by frequency-length product. We use the parameterizable metrics to decide if the block is to be included in the list of blocks for optimizations. For frequency-length product we use:

$$mfl_i(x) = f_i \cdot l_i \tag{1}$$

$$M_{fl}(x) = P_{fl} \cdot \sum_{i=0}^{N} mfl_i \tag{2}$$

$$mfl_i(x) \geq M_{fl} \tag{3}$$

where f_i and l_i are frequency and length (number of cycles) of the basic block i, mfl_i is frequency-length product, P_{fl} is parameter specified by the designer and N is the total number of block in the application. The block is considered for optimizing if inequality 3 is satisfied.

In case of the list sorted by length, we observe the length of the block l_i and

$$M_l(x) = P_l \cdot \max_{i=0}^{N}(l_i) \tag{4}$$

$$l_i(x) \geq M_l \tag{5}$$

where P_l is length parameter specified by the designer. The current block is considered for optimizing if inequality 5 is satisfied.

Given f_i as the frequency of the basic block i,

$$M_f(x) = P_f \cdot \max_{i=0}^{N}(f_i) \tag{6}$$

$$f_i(x) \geq M_f \tag{7}$$

where P_f is frequency parameter. Here also, we include the block in the optimization candidate list if inequality 7 is satisfied. We supply the selected basic blocks to the Histogram Creation step of the AW.

5.2 Histogram Creation

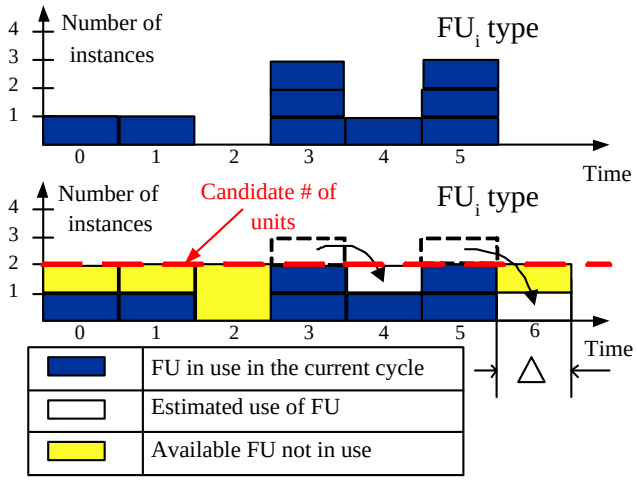

Figure 4. Example of 'Spill' Algorithm.

In this step we create a utilization histogram for each selected basic block for each component type, in case of functional units, and for data ports of the same kind (input or output), in case of the storage units. It is important to group the items of the same kind together in order to easily estimate potential execution and utilization impact when changing the number of instances. The utilization graph is extracted from the schedule generated for the Max Architecture. The example of utilization graph for the functional unit of type i is shown on the top of Fig. 4. The basic block for which the diagram is shown has 6 cycles (0 to 5). It can be seen that no instance of functional unit is used in cycle 2, one instance is used during cycles 0, 1 and 4, and 3 instances are used in the cycles 3 and 5. If we assume that the type and number of instances of all other

resources (memories, register files and its ports, buses and multiplexers) do not change, we can conclude that we need 3 instances of functional unit of type i to execute this basic block in no more than 6 cycles.

5.3 Flattening 'Spill' Algorithm

Let us assume that it is acceptable to trade off certain percentage of the execution time in order to reduce number of components (and therefore reduce area and power and increase component utilization). The designer decides the performance boundary and the desired component utilization and supplies them to the AW. The AW sets the initial value of a candidate number of components to be the largest average number of used instances of a given type for all basic blocks. The goal is to compute how many extra cycles would be required compared to the schedule with Max Architecture and what would their utilization be, if we allocate the candidate number of units. The following algorithm estimates cycle overhead and component utilization.

```
Spill (Histogram, CandidateNumber):
  for all X ∈ cycle
   CycleBudget = CandidateNumber-X.InUse;
    if CycleBudget ≥ 0
      if RunningDemand ≥ 0
       CanFit   =   MIN(CycleBudget, ABS(RunningDemand));
       RunningDemand += CanFit;
      else
       RunninBudget += CycleBudget;
     RunningDemand += CycleBudget;
```

In order to compute execution overhead and utilization we keep two counters: running demand and running budget. Running demand is a counter of operations that are scheduled for the execution in the current cycle on a unit of type i but could not possibly be executed (in the current cycle) with the candidate number of units. For example, in both cycles 3 and 5 in bottom of the Fig. 4 there is one operation that needs to be accounted for by the running demand counter (shown in dashed lines). Running budget counter counts the units that are unused in a particular cycle. In each cycle, we compare the current number of instances with the candidate number. If the current number is greater, the number of 'extra' instances is added to running demand, counting the number of operations that would need to be executed later. On the other hand, if the current number is less then the candidate, we try to accommodate as many operations as possible that were previously accounted for with the running demand counter, modeling the delayed execution. We try to fit in as many operations as possible (represented with 'CanFit' variable in the algo-

rithm) in the current cycle, as shown in the cycles 4 and 6. If there are some unused units left (when the available number of instances is greater than the running demand, like in cycles 0, 2 and 6), the running budget is updated by the number of free units.

We must note that this method does not account for interference while changing the number of instances or ports of other components. The accuracy of a given method will be discussed in Section 6. The presented estimation algorithm uses only statically available information and provides the overhead and utilization for a single execution of a given basic block. In order to be able to compare the resulting performance with the designer's requirements, we incorporate execution frequencies in the estimation.

5.4 Overhead and Utilization Estimation

```
for all C ∈ component
 while C.overhead ≤ Tspec and C.budget ≥ Uspec
  for all C ∈ component
   Spill(Histogram, CandidateNumber)
   C.overhead += B.f * RunningDemand
   C.budget += B.f * EndBudget
  Update(CandidateNumber)
```

The estimation algorithm is shown above. For each of the components, we apply the 'Spill' algorithm to all basic blocks using the largest average number of used units of a particular type across all blocks as a initial candidate number. That way we get 'per block' estimates for the overhead and utilization. Each of these statistics are multiplied by the block frequency (B.f) and accumulated in the global overhead counter (counterpart to the running demand) and global budget counter for a given unit. We also compute the dynamic length of the selected blocks for the Max Architecture by multiplying length by frequency. Having estimates for both new and the baseline architecture, we are able to decide if the candidate number of units will deliver required performance while satisfying utilization constraint. If the candidate number of units does not deliver desired performance, we increment the candidate number and repeat the estimation. If the candidate number of units is suffi cient, we check the utilization, and if it is above the given threshold, we decrement the candidate number and repeat the estimation. In case the algorithm does not converge with respect to the both constraints, we give the priority to performance, and make the decision solely on the overhead.

In the simple case, shown in the the Fig. 4 if the allowed overhead is 20% (i.e. 1.2 cycles for this example) and the desired utilization per unit is 75%, having 3 units would deliver required performance, but would have the units

underutilized. Therefore, having 2 units would be satisfactory solution, with 66% utilization per unit and 17% overhead.

5.5 Allocation and Net-list creation

The allocation slightly differs from the Max Architecture allocation. The storage component allocation is done using the same heuristics with the difference that the required numbers are provided by the previous step of the AW. Previously, during the initial allocation, the operands that were appearing in the code were matched with the components from the library to determine the type of functional unit. Here the functional unit type is inherited from the Max Configuration architecture, and the number of instances is specified by the outcome of the 'Spill' algorithm.

Based on the connectivity statistics, the tool decides to provide full or limited connectivity. The full connectivity scheme is used in Max Architecture as described in Section 4. In limited connectivity scheme, we reduce number of connections from register file's output ports to the source buses, and we connect only one bus to one output port. The tool then connects the provided components according to the scheme provided in Fig. 2. It automatically allocates tri-state buffers and multiplexers as needed for the net-list that is input to NISC compiler.

6. Results

We implemented the Initial Allocation and the Architecture Wizard in C++. For functional simulation of the designs, we use ModelSim SE 5.8c. The experiments were performed on a 1GHz Intel Pentium III running Windows XP. The benchmarks used are *bdist2* (from MPEG2 encoder), *Sort* (implementing bubble sort), *dct32* (from MP3 decoder) and *Mp3* (decoder). The profiling data have been obtained manually. The Architecture Wizard in presented case uses following parameters: Δ = 20%, utilization = 50%, P_{fl} = 0.5, P_f = 0.85, P_l = 0.7.

Table 1. Reduction of number of components in refined design relative to Max Architecture.

Bench.	FUs	Buses	Tri-State	Pipe. Regs
bdist2	50.0	50.0	70.0	42.9
Sort	25.0	50.0	67.7	20.0
dct32	33.3	16.7	52.5	28.6
Mp3	37.5	0.0	40.9	34.6
Avg	36.5	29.2	57.5	35.5

Table 1 gives a comparison of the number of functional units, buses, tri-state buffers, and pipeline registers of refined architecture relative to the Max Archi-

tecture for selected benchmarks. As we can see, the maximum reduction in number of functional units is 50% and the average is 36.5%. The smallest saving is for *Sort*, from 4 to 3 functional units. This is due to the limited number of components in the library: the required operations can be performed with not less than 3 units. The biggest saving is in the case of *bdist2* where half of the underutilized components are removed. The number of buses is reduced by 29.2% on an average. The is no reduction of number of buses in the case of *Mp3*. Due to the application's high parallelism, utilization of all buses if high, and therefore the algorithm does not optimize away any of them. The number of tri-state buffers is reduced by 70% in the best case and 57.5% on an average. The number of pipeline registers is reduced on an average by 35.5%. In the experiments presented here, we do not apply 'Spill' algorithm on pipeline registers, but we automatically inset them as shown in the Section 4. More sophisticated methods of deciding on how to pipeline the given architecture are topic of our research.

Table 2. Measured Δ of refined design and average number of iterations during estimation.

Bench.	Δ[%]	Avg.Iter.	T[min]	LoC
bdist2	18.8	1.8	0.9	61
Sort	0.3	3.9	0.6	33
dct32	19.6	1.2	12.6	1006
Mp3	1.1	1.4	79.5	1389
Avg	9.9	2.1	23.4	0.6

Table 2 shows the performance of resulting designs and illustrates the tool execution characteristics. The first column of the table shows the benchmark, and the next column shows measured Δ overhead in execution cycles of the refined design relative to Max Architecture. It can be seen that all benchmarks satisfy the given constraint of maximum 20% overhead. Benchmark *Sort* experiences negligible run time overhead. This application is sequential and even with the Max Architecture not more than single operation is performed in parallel. The next column shows the average number of iterations per selected basic block per component that the Architecture Wizard performs before converging. For example, on average for a given component in *bdist2* the 'Spill' algorithm will be called 1.8 times to estimate *one* selected basic block. The average number of iterations for all benchmarks is 2.1. The right most column shows the total run time required to generate both Max Architecture and refined architecture and schedule for both. The largest run time is in the order of 80 minutes, making it much faster than any hand written design.

7. Conclusion

We present a C-to-data path technique for designing of custom processors that alleviates the problem of manual architecture design. Our experimental results show that the generated architectures perform within overhead of 19.6% that satisfi es the given performance constraint. Also, the reduction in number of used resources ranges from 29% to 58% on average across all observed components. Our algorithm performs the data path generation within 80 minutes even for large industry size application such as Mp3 decoder. The technique provides fast but effective design alternative making it possible for designers to create and evaluate many different design alternatives in less time than required for single custom design iteration. In future, we plan to add more capabilities to the automatic selection algorithm and to improve the quality of generated architectures by implementing automatic pipelining, forwarding and automatic customizing of memory hierarchy. Our current efforts are directed to overall area reduction by using multiplexer-based instead of bus-based interconnect and by minimizing the area of functional units.

Acknowledgments

The authors wish to thank Mehrdad Reshadi and Bita Gorjiara for compiler support, Verilog generator and stimulating discussions that have improved this work. We would also like to thank Pramod Chandraiah for providing the Mp3 source code.

References

Automated Configurable Processor Design Flow (2005). Automated Configurable Processor Design Flow, White Paper, Tensilica, Inc. http://www. tensilica.com/pdf/Tools_white_paper_final-1.pdf January 2005.

B. Landwehr, P. Marwedel, and R. Dømer (1994). OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. In *Proc. European Design Automation Conference*, pages 90–95, Grenoble, France. IEEE Computer Society Press.

Brewer, F. and Gajski, D. (1990). Chippe: A system for constraint driven behavioral synthesis. *IEEE Trans. on Computer-Aided Design*.

Devadas, S. and Newton, R. (1989). Algorithms for hardware allocation in data path synthesis. *IEEE Trans. on Computer-Aided Design*.

Diamond Standard Processor Core Family Architecture (2006). Diamond Standard Processor Core Family Architecture, White Paper, Tensilica, Inc. http://www. tensilica.com/pdf/Diamond WP.pdf, October 2006.

Gajski, Daniel (October 2003). Nisc: The ultimate reconfigurable component. Technical report, Technical Report TR 03-28, University of California-Irvine.

Goodwin, David and Petkov, Darin (2003). Automatic generation of application specific processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*.

Gutberlet, P., Mu̇ller, J., Kramer, H., and Rosenstiel, W. (1992). Automatic module allocation in high level synthesis. In *Proceedings of the Conference on European Design Automation (EURO-DAC '92)*, pages 328–333.

Marwedel, P. (1993). The MIMOLA system: Detailed description of the system software. In *Proceedings of Design Automation Conference*. ACM/IEEE.

Paulin, P.G. and Knight, J.P. (1989). Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Reshadi, M. and Gajski, D. (2005). A cycle-accurate compilation algorithm for custom pipelined datapaths. In *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

Reshadi, M., Gorjiara, B., and Gajski, D. (2005). Utilizing horizontal and vertical parallelism with no-instruction-set compiler for custom datapaths. In *In Proceedings of International Conference on Computer Design*.

Stretch. Inc.: S5000 Software-Configurable Processors (2005). Stretch. Inc.: S5000 Software-Configurable Processors http://www.stretchinc.com/products/ devices.php.

Tensilica: Xtensa LX (2005). Tensilica: Xtensa LX http://www.tensilica.com/products/xtensa_LX.htm.

Trajkovic, Jelena, Reshadi, Mehrdad, Gorjiara, Bita, and Gajski, Daniel (2006). A graph based algorithm for data path optimization in custom processors. In *Proceedings of 9th EUROMICRO Conference on Digital System Design*, pages 496–503. IEEE Computer Society.

Tsai, Fur-Shing and Hsu, Yu-Chin (1992). STAR: An automatic data path allocator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(9):1053–1064.

Tseng, C. and Seiwiorek, D.P. (1986). Automated synthesis of data paths in digital systems. *IEEE Trans. on Computer-Aided Design*.