A Graph Based Algorithm for Data Path Optimization in Custom Processors

Jelena Trajkovic, Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski
Center for Embedded Computer Systems
University of California Irvine
Irvine, CA 92697-3425, USA
jelenat, reshadi, bgorjiar, gajski@cecs.uci.edu

Abstract

The rising complexity, customization and short time to market of modern digital systems requires automatic methods for generation of high performance architectures for such systems. This paper presents algorithms to automatically create custom data path for a given application that optimizes both resource utilization and performance. The inputs to the architecture generator include application source code, operation execution frequency obtained by the profile run and a component library (consisting of ALUs, busses, multiplexors etc.). The output is the application specific data path specified as the set of resource instances and their connections. The algorithm starts with a dense architecture and iteratively refines it until an efficient architecture is derived. The key optimization goal is to keep performance within given boundaries while maximizing resource utilization. Our experimental results show that generated architectures are comparable to manual designs, but can be obtained in a matter of few seconds, thereby leading to significant productivity gains.

1 Introduction

System on Chip (SoC) design has fueled a need for specialized processors for different application domains. Full HW design is too expensive and rigid for tuning the design later for debugging and feature extension. General purpose embedded processors are often too slow and power hungry to be chosen as an implementation alternative for niche applications. Although custom processors are a better fit for SoCs, they still need to be adapted to run the chosen application efficiently. Such adaptation includes the selection of an appropriate data path architecture. Manual design of data path can be time consuming and error-prone. 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/resource constraints.

We follow a design methodology for custom processors that separates the allocation of architectural resources and their connections from the scheduling of control words that drive that data path. The reasoning behind this separation of concerns follows from our past experiences in custom processor design. Since architecture selection and scheduling are inherently difficult problems, performing them together leads to excessive run times and typically sub-optimal results. The difficulty of doing scheduling and architecture selection simultaneously has been a significant factor in the limited acceptance of high level synthesis tools.

Figure 1 shows our custom processor design methodology. The application code is first scheduled in an As Late As Possible (ALAP) fashion. After an application's requirements have been derived from ALAP scheduler, they are used for allocation of the data path components. The only limitation here is imposed by the components available in the components library. The resulting architecture is evaluated and refined:the estimation of the execution time and utilization for different number of components is done until the resulting architecture satisfies given criteria.

The paper is organized as follows. We compare our approach to related work in Section 2. In Section 3 we describe proposed methodology and the tool flow that implements it. In Section 4 we present details of initial allocation algorithm and the heuristics used to match application resource requirement to the components available in the library. We describe refinement and evaluation algorithms in Section 5. The results are presented in Section 6. We summarize and point out to the future work in Section 7.

2 Related Work

There is large body of work in high level synthesis where the problems of scheduling, allocation and binding are been solved together ([2, 9, 3, 7, 6, 5, 1, 8, 12]). For example, the technique presented in [1] uses integer linear programming. Linear programming is, in practice, applicable to fairly small problems (the time to search for the optimal

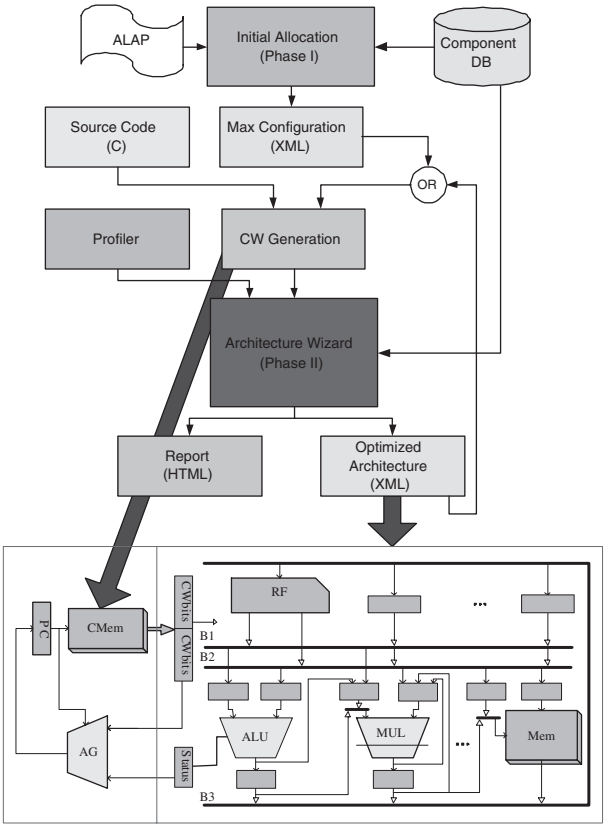

Figure 1. Custom Processor Design Methodology.

solution increases exponentially with the design size) and known to be NP-complete. To reduce the computation time comparing to linear programming approach, Palulin et.al. in [8] used force directed scheduling (FSD). This algorithm performs the scheduling by uniformly distributing the operations of the same type across the available control steps. However, the FSD does not always produce an optimal solution. Tseng et. al. in [12] use graphs, where nodes represent elements to be assigned to hardware and edges denote that two notes may share the same hardware resource. The allocation is then reduced to finding a maximal cliques in the graphs (NP-complete). Marwedel in [6] uses branch-and-bound technique to search for the optimal solution for allocation problem. Since this approach is potentially exponential, the heuristics may be used to limit the search space, but the resulting allocations is not guaranteed to be optimal.

All of the prior work performs allocation, scheduling and binding together. The main difference of our approach is that we separate allocation from scheduling and binding, which reduces the problem size and significantly decreases the run-time.

3 Methodology

We propose a custom processor design methodology for the No-Instruction-Set Computer (NISC). NISC style processor differs from both well known processor types: CISC and RISC. CISC processor uses complex instruction set where instructions usually take several cycles to execute and are stored in the micro program memory. The CISC concept allowed for emulation of any instruction set and creation of specialized instructions, but failed to be efficiently applied to the pipelined data paths. On the other hand, RISC instructions are simple and execute in one cycle (on non-pipelined architecture), allowing efficient pipelining. The micro program memory is replaced by the decoding stage. However this processor type have fixed instruction set which can not be easily modified. NISC completely removes the decoding stage and stores the control words in the program memory. The NISC compiler ([10, 11]) compiles the application directly onto given data path, creating a set of control signals (called control word) that drive 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 Figure 1. It consists of 2 phases: *Phase I* called *Initial Allocation* and *Phase II* called *Evaluation* that is implemented by the *Architecture Wizard* tool. In the Initial Allocation phase, we use the schedule information (ALAP schedule) 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 Configuration.

The Max Configuration 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 given back to the Architecture Wizard (performing the Evaluation phase). 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 net-list of the optimized architecture and the report in the human readable format.

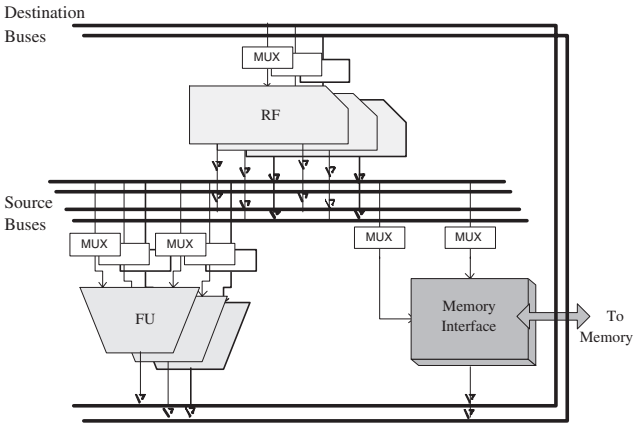

Figure 2. Components and connections.

4 Initial Allocation

We start by defining maximal requirements of a given application from the application's ALAP schedule, where the ALAP is made assuming infinite resources available. In general, any schedule may be input to our tool. 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 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. For example, if the ALAP schedule assigns 5 additions in the same cycle, then 5 unit that perform addition will be allocated, together with sufficient number of busses, register files and memories to provide for reading and writing of all operands. However, since the resources are selected from the choices in CL, the resulting architecture depends on the number and variety of components in the CL.

To ensure that the interconnect is not a bottleneck in the Max Configuration, we perform a greedy allocation of connection resources (Figure 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.

4.1 Component Selection

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. Therefore, the functional units may also be indexed by the the operation, allowing us to allocate the instance of functional unit for the given operation. The library also carries a link to the synthesizable RTL Verilog code for each component. The designer may choose to use only available components or to extend the library. For adding the new component, the aforementioned information needs to be added to the CL, and the corresponding Verilog code for the unit needs to be supplied to the NISC compiler.

For component allocation, we have derived several heuristic that measure how well the selected components match the given requirements. We start by allocating the register file. We use the following formula to evaluate each component:

$$H\_rf(x) = 2 * (x\_in - rq\_in) + (x\_out - rq\_out), \quad (1)$$

where x is the candidate component form the library, and x_in and x_out are number of input and output ports of the component x. rq_in and rq_out are number of required inputs and outputs respectively. Required number of outputs correspond to the number of source operands computed by the Allocator, and number of inputs correspond to the number of destinations. The heuristic is chosen to give priority to the input port requirement because we wanted to allow storage of as many results as possible. The candidate component, with the smallest value of the function H_rq is chosen, and allocated to the data path.

Once the register file is selected, the source and the destination buses are allocated. The number of source buses for the Max Configuration is equal to the number of register file's output ports, if that number is odd, or (number of register file's out ports + 1) otherwise. The number of destination buses solely depends on the number of the register file's input ports.

As for the memory interface allocation, we first consider number of sources and destination buses and chose the maximum of them to serve as the required number of ports rq_mi. We compute the value of H_mi for each candidate x component according to:

$$H\_mi(x) = x\_in - rq\_mi, \quad (2)$$

where x_in is the number of input ports of memory interface. The component with the minimum value of the heuristic is selected. In the corner case, where rq_mi is less (or greater) than any of x_in the memory interface with the minimum (or maximum) number of ports is chosen.

While allocating functional units, we choose the type of unit that alongside the given operation, performs the largest

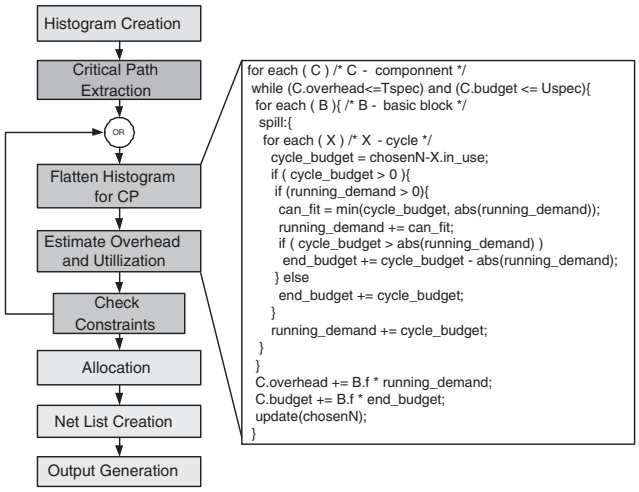

Figure 3. Architecture Wizard - top level view.

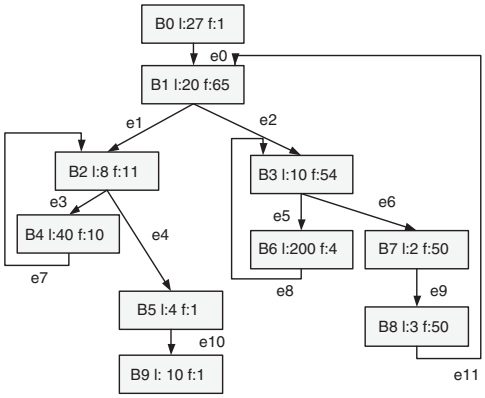

Figure 4. Example of the Original Graph for a Given Application

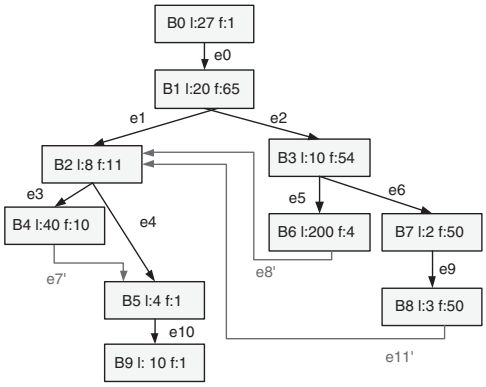

Figure 5. Removing Backward Edges: Example of Application's DAG.

number of operations. That way, we prevent allocation of too many units and allow the Phase II 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 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 to exceed the number of source buses.

5 Estimation and Refinement

Phase II of the Architecture Selection tries to reduce number of used resources to create the final design that matches given performance and utilization goals. The source code is first compiled using the Max Configuration specified by the Initial Allocation. 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 Architecture Wizard.

Figure 3 shows the tool flow. We start by creating the histogram for each functional unit type or for each input or output port of the storage unit. It is necessary to consider the utilization of all components of the same type in order to apply 'Spill' (estimation) algorithm described in Section 5.2. Next, we select the part of the application that we want to optimize (*Critical Path Extraction*).The algorithm used for this step will be presented in Section 5.1. For the selected path, we try to estimate the number of instances of the chosen component that will keep the execution within the given boundaries and utilization. This loop runs until both the performance and utilization constraints are satisfied for the given component. Once we have reached the decision about the optimal number of components, we go through the allocation, net-list creation and output generation (Section 5.3).

5.1 Critical Path Extraction

The goal of this phase of the Architecture Wizard is to select the critical path of the application. The question is how to decide which basic blocks are the most critical or the most promising for optimization. For reducing overall execution time we need to consider delays from start of the application till its end. We define the critical path of the application as the sequence of basic blocks from start to the end that contribute the most to the execution time. For a given application and its Max Configuration architecture, we have two sets of data, namely the length of each basic block in cycles (derived from schedule) and the execution frequency of each basic block during profiling.

Let us represent the application using a graph, where

each basic block i is represented by node B_i and there is an edge between nodes B_i and B_j if basic block j is a possible successor of block i. A simple example of such a graph is shown in Figure 4. Each node of this graph is annotated with the basic block's length ('l') and frequency ('f'). In order to find where the given application spends most of its execution time, we need to find the longest possible path from the starting to the ending basic block: in this case from block $B1$ to block $B8$ as shown in Figure 4. Each block has a weight attached to it. The weight may be chosen to be the block's length, frequency or the product of length and frequency. The problem is to find the path where the sum of the weight of the nodes in that path is maximum. This path is our critical path that is chosen for optimization. We propose a set of transformations that reduce the critical path problem for a (possibly cyclic) graph to the well known shortest path problem. The main transformations are:

1. Creation of direct acyclic graph (DAG) form a given graph
2. Creation of a dual graph (edge-node substitution)
3. Weight computation

The original graph for a given application may potentially contain cycles, like one in Figure 4. In order to apply one of the shortest path algorithms we need to remove the backward pointing edges (e_7, e_8 and e_{11} in Figure 4). Those edges can not simply be erased, because there would be no way to find out all other potential paths form the given loop to the end of the application. Therefore we re-link the backward pointing edges in such matter that we keep the path information, using the following algorithm:

X: node
$X.pred$: set of nodes preceding X
e_i $(X \rightarrow Y)$: edge i from node X to Y
for all $e_i(X \rightarrow anynode$ **do**
  if e_i $(X \rightarrow X)$ **then**
    remove e_i
  end if
  if $(e_i(X \rightarrow Y))$ and $Y \in X.pred$ **then**
    mark e_i for removal
    if $\nexists$ e_j $(X \rightarrow N)$ where $N \notin X.pred$ **then**
      find e_k $(Y \rightarrow Z)$ such that
      $(\nexists$ e_m $(Z \rightarrow X)$ and $(\nexists$ e_p $(Z \rightarrow Q)$ where $Q \in X.pred))$
      create e_q $(X \rightarrow Z)$
    end if
  end if
end for

We examine each outgoing edge of every node. In case the edge points to the source node itself, we remove the edge. Otherwise, we compare the destination with the set of predecessors of the source node. In case the destination is in the set of predecessors, we mark the edge for removal and try finding the candidate replacement destination. To find the replacement destination, we explore all edges of the original destination node trying to find one that points to the node other than the original source or any of its predecessors. Once such a node is found, we remove the marked edge and add the one from the original source to the candidate destination. The resulting graph is a direct acyclic graph *DAG*.

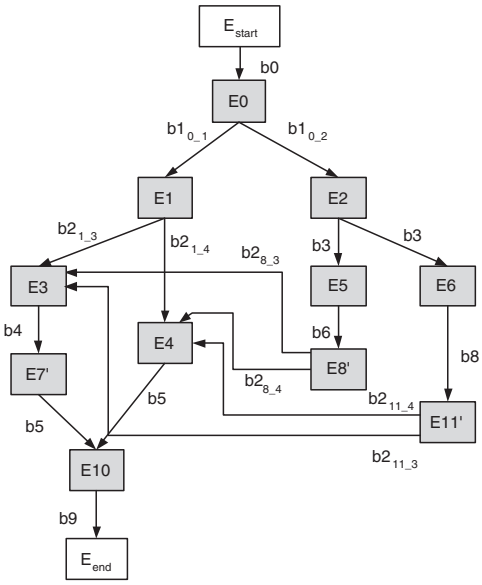

Figure 6. Dual Graph of the Given Application.

The next step is the creation of the dual graph in which edges become nodes, and nodes are translated into edges as described by the following algorithm:

Create E_{start} to be source to the edge b_0
for all $e_x \in DAG$ **do**
  Create a node $E_x \in Dual$
end for
for all $B_y \in DAG$ **do**
  for all incoming e_i $(P \rightarrow Y) \in DAG$ **do**
    for all outgoing e_o $(Y \rightarrow Q) \in DAG$ **do**
      Create edge b_y $(E_p \rightarrow E_q) \in Dual$
    end for
  end for
end for
Create E_{end}

We start by allocation a starting node E_{start}. For each edge e_x in the given DAG a node E_x is created in the dual graph. Each edge B_y from the DAG gets translated into $n \times m$ edges, where n is number of incoming and m is

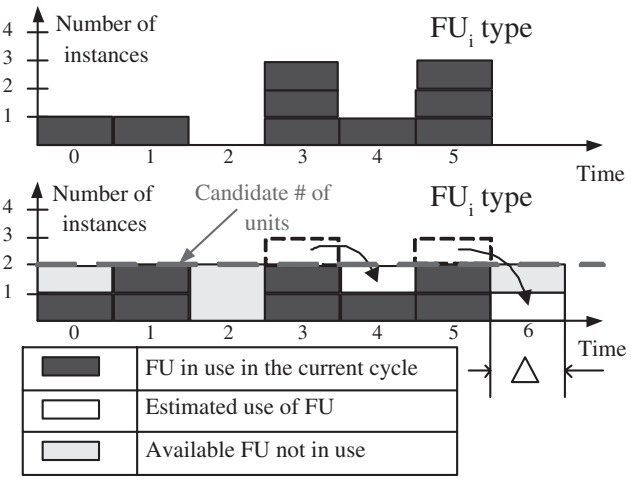

Figure 7. Example of 'Spill' Algorithm.

the number of outgoing edges of B_y. Finally, we create an ending node E_{end} such that there is the destination node for the edge that corresponds to the last block in the DAG ($B9$ in Figure 5). The resulting dual graph is shown in the Figure 6.

The last step is transforming the longest path to the shortest path problem. Here we simply compute the new weights for each edge: one may use $1/w_i$ or $W_{max} - w_i$ where W_{max} is the maximum of all weights in the graph. We use topological ordering, but any algorithm for finding the shortest path can be applied to this graph. More on the graph algorithms can be found in [4].

5.2 Spill Algorithm

The task of the Architecture Wizard is to come up with the data path architecture that will deliver the performance within given limits while minimizing the number of the components and connections used. The optimization starts by creating the utilization histogram 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 example of utilization graph for the functional unit of type i is shown on the top of Figure 7. 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 multiplexors) 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. Let's 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 is responsible for deciding about the performance boundary and the desired component utilization.

The example of 'Spill' algorithm is shown in the bottom of Figure 7. We start with a candidate number of components: in our case that is the largest average number of used instances of a given type for all basic blocks. In general, any other number, including maximum and minimum number of used units may be used. We are interested in computing how many extra cycles would be required compared to the schedule with Max Configuration architecture and what would their utilization be, if we allocate the candidate number of units.

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 Figure 7 there is one operation that needs to be accounted for by the running demand counter (shown in dashed lines). Running overhead 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 running demand counter, modeling the delayed execution. We try to fit in as many operations as possible 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.

Note that this is only estimation, not a schedule. 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.

The estimation algorithm is shown on the right hand side of the Figure 3.For each of the components, we apply the

Table 1. Comparison of Max Configuration and Refined design.

	FUs		Buses		Tri-State	
Bench.	MC	R	MC	R	MC	R
bdist2	6	4	6	5	40	19
OnesCounter	3	2	6	5	34	17
Sort	4	3	6	5	36	18
dct32	6	4	6	5	40	19

'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 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 Configuration 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 sufficient, 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 Figure 7 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% of utilization per unit and 17% of overhead.

5.3 Allocation, Net-list creation and Output Generation

The allocation only slightly differs from the Max Configuration allocation. The storage component allocation is done using the same heuristics as described by Equations 1 and 2. The difference is that here the required numbers are provided by the estimator. The number of bus instances is also decided by the estimation algorithm. 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 'Spill' algorithm.

After observing the connectivity statistics, the tool decides to provide full or limited connectivity. The full connectivity scheme is used in Max Configuration 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 Figure 2. It automatically allocates tri-state buffers and multiplexors as needed creating the internal net-list of components. The internal net-list data structure is used to generate the output file according to the format required by the compiler. The tool also outputs a simplified description of optimized architecture in human readable HTML format.

Table 2. Performance comparison of Max Configuration and Refined design.

Bench.	Overhead[%]	Avg. Iterations	T[s]
bdist2	32.1	2.8	0.05
OnesCounter	11.9	1.4	0.05
Sort	0.6	2.9	0.06
dct32	31.1	1.4	0.48

6 Results

We implemented the Initial Allocation and the Architecture Wizard as the part of the NISC tool flow. We use C++ and the Microsoft Visual C++.NET as the implementation platform. For the functional simulation of the designs, we use ModelSim SE 5.8c. The parameters for the Architecture Wizard were selected to be 20% for the allowed execution overhead and 75% for the desired component utilization. The experiments were performed on a 1GHz Intel Pentium III running Windows XP. The benchmarks used are *bdist2* (from MPEG2 encoder), *OnesCounter*, *Sort* (implementing bubble sort) and *dct32* [1] (function from MP3 encoder). In case of the benchmarks *bdist2* and *dct32* the execution time does not depend on the input data set. On the other hand, for *OnesCounter* and *Sort* we used the input data that makes them run in the worst case execution time. *OnesCounter* operates on maximum 32-bit number, and *Sort* operates on 100 elements, with input data in the reverse order of the desired. The profile information are obtained manually.

[1] We would like to thank Pramod Chandraiah for providing the source code.

As described in Section 3 we first generate the Max Configuration. This is followed by the refinement phase implemented by the Architecture Wizard. Table 1 gives a comparison of the number of functional units, buses and tri-state buffers between the Max Configuration(MC) and the final refined architecture(R) for selected benchmarks. As we can see, the maximum reduction in number of components is 33% and the average is 31%. The smallest saving is for the *Sort*, because of the nature of the application: the application actually needs to use 3 different functional units, and hence the number of instances can not be reduced further. The number of buses is reduced by 16% in all cases where the number of tri-state buffers is reduced by 52% in the best case and 51% on the average.

Table 2 compares the performance of both designs and illustrates the tool execution characteristics for the same set of benchmarks as in Table 1. The first column of the table shows the benchmark, and the next column shows the overhead in execution cycles between Max Configuration and the refined design. This number is shown as percentage of the execution on the Max Configuration. It can be seen that the first tree benchmarks satisfy the given constraints. The maximum deviation from the specified overhead is for the *dct32*: for this benchmark, the number of buses fails to provide required number of source and destination operand. This example points our future research to incorporating the interference between components in our estimation algorithm. The next columns 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 2.8 times to estimate *one* selected basic blocks. The average number of iterations for all benchmarks is 2.1. The right most column (T) is total time required to run the refinement. The *dct32* has the highest runtime amongst selected benchmarks, in spite of relatively small average number of iterations. This is because it has large basic blocks and majority of time is spent in algorithm, not in iterations. The average run time is 19 ms.

7 Conclusion

As the complexity of digital systems continues to grow, we are faced with the challenge of designing such systems within shrinking time to market. We presented a method for automatic generation of data path for custom processors that alleviates the problem of manual architecture design. We presented our work in the context of a novel custom processor design methodology that separates data path design from code scheduling. This is in contrast to previous efforts in the field that have been riddled with the vicious circle of optimizing both data path and schedule together. Our experimental results demonstrate the feasibility and efficiency of the data path selection algorithm as compared to a manual design. In the future, we plan to add more capabilities to the automatic selection algorithm to account for more sophisticated architectures including pipelining, chaining and forwarding. We also plan to improve our heuristics to further reduce the overall area of functional units.

References

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

[2] R. K. Brayton, R. Camposano, G. De Micheli, R. Otten, and J. van Eijndhoven. The yorktown silicon compiler system. In D. D. Gajski, editor, *Silicon Compilation*, pages 153–203 or 204–311. Addison-Wesley, 1988.

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

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edn.* Cambridge, MA: MIT Press., 2001.

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

[6] P. Marwedel. The MIMOLA system: Detailed description of the system software. In *In Proceedings of Design Automation Conference*. ACM/IEEE, June 1993.

[7] N. Parkand and A.C.Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Trans. on Computer-Aided Design*, Mar. 1988.

[8] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, June 1989.

[9] P. G. Paulin, J. P. Knight, and E. F. Girczyc. HAL: A multi-paradigm approach to automatic data path synthesis. In *In Proceedings of Design Automation Conference*, pages 263–270. ACM/IEEE, July 1986.

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

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

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