Cluster Assignment for High-Performance Embedded VLIW Processors

VIKTOR S. LAPINSKII, MARGARIDA F. JACOME,
and GUSTAVO A. DE VECIANA
The University of Texas at Austin

Clustering is an effective method to increase the available parallelism in VLIW datapaths without incurring severe penalties associated with a large number of register file ports. Efficient utilization of a clustered datapath requires careful binding/assignment of operations to clusters. The article proposes a binding algorithm that effectively explores trade-offs between in-cluster operation serialization and delays associated with data transfers between clusters. Extensive experimental evidence is provided showing that the algorithm generates high quality solutions for representative kernels, with up to 33% improvement over a state-of-the-art binding algorithm.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation*; *Compilers*; *Optimization*

General Terms: Algorithms

Additional Key Words and Phrases: Operation binding, clustered VLIW datapaths, embedded systems, embedded processors, partitioning

1. INTRODUCTION

A significant segment of embedded multimedia applications exhibits high instruction-level parallelism (ILP) in the most time-consuming inner loop kernels. Very Large Instruction Word (VLIW) processors provide a means to efficiently exploit such ILP. A "simple" VLIW datapath may consist of a centralized register file (RF) with several functional units (FUs) connected to it through dedicated ports (see Figure 1). With a sufficient number of FUs, a compiler may be able to utilize all the available static ILP present in a given kernel. However, as the number of functional units (and thus register file ports) increases, such "centralized" architectures may become prohibitively costly in terms of clock rate, power, area, and overall design complexity [Rixner et al. 1999].

This work is supported in part by an NSF ITR Grant ACI-0081791, an NSF Grant CCR-9901255, and by Grant ATP-003658-0649 of the Texas Higher Education Coordinating Board. This is an extended version of a paper presented by the same authors at the 38th International Conference on Computer-Aided Design.

Authors' address: Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712 ; e-mail: {lapinski,jacome,gustavo}@ece.utexas.edu.

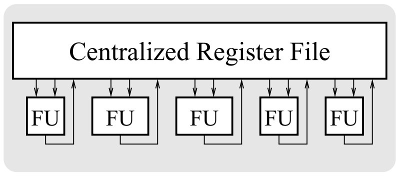


Fig. 1. A simplified model of a centralized VLIW datapath.

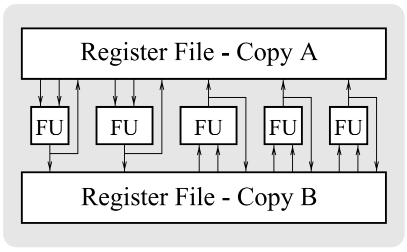


Fig. 2. Decreasing the number of ports by register file replication.

A group of approaches relies on restricting the connectivity between FUs and registers in order to control the penalties associated with numerous RF ports, while still providing a sufficient number of functional units to exploit the available ILP. This is implemented by splitting the centralized RF into several register files with fewer ports in each one.

One such approach (see, e.g., White and Dhawan [1994] and Ebcioğlu et al. [1998]) is based on the fact that the number of outputs of a functional unit (typically 1) is smaller than the number of inputs (typically 2). The idea is to write each result to several register files which makes it possible to reduce the number of read ports in each RF as illustrated in Figure 2.

A more radical solution, however, is to structure a VLIW datapath into a set of clusters (see Figure 3), each cluster containing a set of functional units connected to a local register file. This approach was used in the past [Colwell et al. 1990] and has been adopted in several recent industry products [Texas Instruments 2000; Analog Devices 2001; Basoglu et al. 2000] and research projects [Rau et al. 1998; Hanno and Devadas 1998; Fritts et al. 1999; Faraboschi et al. 2000; Kailas et al. 2001]. In contrast with centralized architectures, clustered architectures scale well because they use smaller register files with fewer ports. Giving up full connectivity also helps to decrease the complexity of the bypassing structure. In such clustered architectures, however, additional explicit data transfer operations may be required to move data from one cluster to another (see Figure 3), which may lead to increased schedule latency and energy consumption.

This article presents an algorithm for binding/assigning operations in a dataflow graph[1] (DFG) to the datapath clusters, so as to minimize latency

[1]The dataflow graph represents a time-critical inner kernel, which may be a basic block or an equivalent region, such as hyperblock, superblock, and the like.

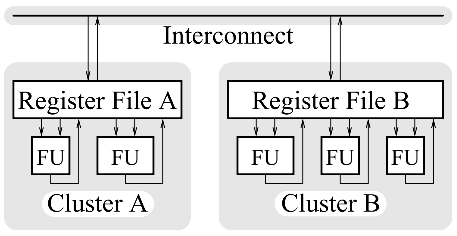


Fig. 3. An example of clustered datapath.

(primary figure of merit) and data transfers (secondary figure of merit). The algorithm was originally designed to work with VLIW datapaths, specialized to execute time-critical kernels of embedded applications [Lapinskii et al. 2002]. In that context, it was important to verify its performance on a wide variety of datapath configurations, including number of clusters, number and types of FUs in each cluster, FU and bus latencies, and so on (see Section 4).

Note that some approaches (see Section 5) unify/integrate the binding, scheduling, and register allocation problems. Due to the high complexity of the joint problem, these methods are necessarily very greedy. By contrast, and similarly to other approaches targeting clustered machines (most notably, Desoli [1998] and Faraboschi et al. [1998]), we assume a phasing where binding (the problem addressed in this article) precedes final scheduling and register allocation. Indeed, as mentioned above, clustered datapaths are typically designed to support high instruction-level parallelism. The decrease in the number of RF ports achieved through clustering allows one to increase the size of local register files with a significantly reduced impact on delay/power. Thus, in a clustered machine with a proper "balance" of resources (functional units and registers), spills should be rare and not significantly affect performance. In other words, for this class of processors, a high-quality partitioning of operations across clusters (possibly evaluated by comparing estimates of schedule latencies) should not be significantly affected by spills due to register shortage.

Our algorithm comprises the phases: (1) generation of an initial binding solution (B-INIT), and (2) iterative improvement[2] of the initial solution (B-ITER). We note that the fast initial binding algorithm (used in Phase 1) already delivers good quality results (see Section 4), and may thus be used (alone) when compilation time is very critical. The second, iterative improvement, phase is designed to deliver maximum quality results when code performance is the major goal. In Section 4 we report improvements of up to 25% (for B-INIT) and up to 33% for (B-ITER) over PCC [Desoli 1998; Faraboschi et al. 1998], a state-of-the-art binding algorithm.

In addition to its obvious relevance to code generation, (see Jacome and de Veciana [2000]), our binding algorithm has proven to be a key component of several optimization algorithms addressing other compilation/synthesis tasks.

[2]Note that the iterative improvement phase uses a fast scheduling algorithm to estimate the quality of bindings as described in section 3.2.1.1.

These include a register and code size-sensitive software pipelining algorithm for clustered VLIW machines [Akturan and Jacome 2001] and a design space exploration framework for clustered VLIW datapath configurations [Lapinskii et al. 2002].

The rest of this article is organized as follows. In Section 2 we introduce the models used in our approach. Section 3 details the proposed binding algorithm. We present experimental results in Section 4, discuss previous work in Section 5, and conclude in Section 6.

2. DATAPATH AND DATAFLOW MODELS

2.1 Datapath

We model the datapath of a VLIW processor as a set of clusters CL connected through an interconnect resource, for example, a bus (see Figure 3). Each cluster $c \in CL$ comprises a register file and a collection of functional units. Each functional unit can read up to two operands and write one result to the register file, through dedicated RF ports.

Each functional unit in the datapath belongs to a corresponding *functional unit type* (resource type) t. The number of functional units of type t in cluster c is denoted by $N(c, t)$. The total number of FUs of type t in a datapath is:

$$N(t) = \sum_{c \in CL} N(c, t).$$

For generality, we consider the bus to be a resource of type BUS, and denote the bus capacity (i.e., the number of simultaneous inter cluster data transfers that the bus can perform) by $N(BUS)$ or N_B for short. Other equivalent interconnect structures (e.g., a crossbar) can be modeled similarly.

Although the algorithms described in this article are general in terms of the number of FU types they can handle, only two types (ALU and multiplier) are used in our examples, so as to facilitate the presentation of results (see Tables II, III, and IV). Every operation v in the dataflow belongs to an operation type—*optype*(v). In our model, each operation type p is associated with one functional unit type *futype*(p) (e.g. “subtraction” is performed on ALUs) and thus, the set of functional unit types partitions the set of operation types.[3] Note that the data transfer (move) operation type is associated with the resource type BUS; that is, *futype*(*move*) $=$ BUS.

As mentioned in Section 1, the proposed binding algorithms do not consider register allocation when estimating the quality of candidate solutions, and thus register file sizes are not included in the model. We further discuss this in Section 4 (experimental validation).

[3]As mentioned above, our algorithm was primarily developed to work with specialized embedded VLIW machines. The above “partition” assumption reflects that initial aim, and may not hold for certain readily available VLIW processors. Specifically, using this algorithm without modifications for VLIW machines with a significant degree of functionality overlapping among FU types may lead to poor quality results. We are currently working on an extension allowing the algorithm to handle datapaths with such overlapping.

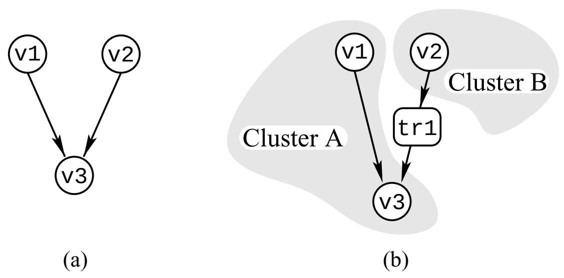


Fig. 4. Dataflow model: (a) original DFG; (b) bound DFG.

2.2 Dataflow

Our model utilizes a dataflow graph (DFG) representation of the kernels of interest. A DFG is a direct acyclic graph $DAG = (V, E)$. The set of vertices V represents operations and the set of edges $E \subset V \times V$ models data dependencies between operations.

A DFG can assume two forms: the original and the bound, as illustrated in Figures 4(a) and (b), respectively. The latter includes the data transfer operations necessary to deliver data objects from the datapath clusters where they are produced to the ones where they are consumed (see, e.g., the data transfer $tr1$ inserted between operations $v2$ and $v3$ in Figure 4(b).

The binding function is denoted by $bn(v)$. An operation v can be *bound* to a cluster c if c has a functional unit supporting that operation type. In other words, $bn(v) = c$ is admissible if $N(c,\ futype(p)) > 0$ for $p = optype(v)$. The set of clusters supporting an operation v of type p is called the *target set* for that operation: $TS(v)$. The binding problem can thus be formulated as the selection of a cluster c in the target set $TS(v)$, for each operation $v \in V$ in the original DFG.

2.3 Timing Model

In general, each pair (p, t) of operation type p (including moves) and resource type t supporting p, has a corresponding *latency* $lat(p, t)$, which is defined as the number of clock cycles needed to produce the result at a specified location. In our model, resources can be pipelined.[4] For a pipelined resource of type t executing an operation of type p, we define a *data introduction interval* as the number of clock cycles after which the resource is ready to start a new operation, and denote it by $dii(p, t)$.[5] Since in our model resource types partition the set of operation types, we use a simplified notation for latency ($lat(p)$) and data introduction interval ($dii(p)$). For convenience, we use the same notations $lat()$ and $dii()$ for operations and their types: $lat(v) = lat(optype(v))$ and $dii(v) = dii\ (optype(v)) = dii\ (futype(v))$.

[4]Our model of pipelining can be applied to both FUs and buses, yet pipelining the latter may be too expensive.

[5]When an operation is executed on a nonpipelined resource, $lat(p, t) = dii(p, t)$.

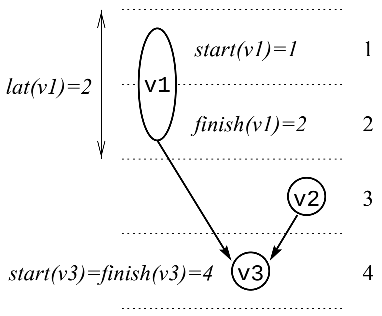


Fig. 5. Scheduled operations.

The start and finish times of a (scheduled) operation v are denoted by $start(v)$ and $finish(v)$, respectively (see Figure 5). Their relation to the latency of v is the following:

$$finish(v) - start(v) = lat(v) - 1.$$

Note that $start(v)$ is also referred to as the "scheduling step" of v. The constraints imposed by data dependencies E on start and finish times of operations are as follows. Given an edge (v_i, v_j), where v_i or v_j can also be data transfer operations, $start(v_j) > finish(v_i)$.

The *schedule latency* L denotes the number of clock cycles necessary to complete the execution of all operations in the DFG:

$$L = \max_{v \in V} finish(v).$$

The functions $alap(v)$ and $asap(v)$ denote the "as late as possible" and "as soon as possible" scheduling steps of v, respectively. [de Micheli 1994] The latter is defined in the context of a *target latency* L_{TG}; specifically, $alap(v)$ represents the latest possible scheduling step of v, such that a valid schedule that completes in L_{TG} cycles is still feasible on a datapath with infinite resources.

For a given target latency L_{TG}, the mobility $\mu(v)$ of an operation v is defined as $\mu(v) = alap(v) - asap(v)$. Note that mobility of operations in the original and the bound DFG may differ because of required data transfers.[6]

3. BINDING ALGORITHM

The binding algorithm proposed in this article consists of two phases. The phase B-INIT, discussed in Section 3.1, performs a coarse DFG partitioning aimed at increasing the parallelism in the final schedule, as well as minimizing the number of data transfer operations. Despite its low complexity, the algorithm used in this phase delivers very good results. Still, if a better solution is needed, the second phase, B-ITER, discussed in Section 3.2, delivers better quality solutions, at the expense of increased time complexity. The top-level algorithm starts by invoking the initial binding phase (Section 3.1) a number of times, varying several parameters, as discussed in Sections 3.1.3 and 3.1.4. It then passes the

[6]Mobility of data transfer operations in the bound DFG is defined similarly to that of regular operations.

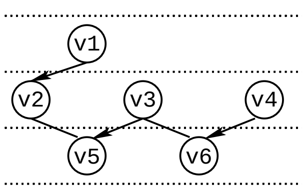


Fig. 6. Illustration of binding order.

best result obtained by the initial binding to the iterative improvement phase (Section 3.2).

3.1 Initial Binding Phase

The initial binding phase is based on an efficient greedy algorithm that includes the key elements: (1) a ranking function, used to determine the order in which nodes are considered for binding; and (2) a cost function that drives the actual binding process. A good ordering ensures that higher priority is given to the most critical binding decisions (i.e., the most difficult/constrained operations are bound first), and the most "flexible" nodes are left for later steps. The cost function should adequately predict the "global" effect of such "incremental" binding decisions, and at the same time be computationally inexpensive.

3.1.1 *Ordering.* One of the simplest ways to order operations is to use mobility (see Section 2.3) as the ranking function. The rationale behind such an ordering is that operations with smaller mobility have fewer alternatives for scheduling, and thus should be considered first. Unfortunately, with this ordering the algorithm tends to traverse the DFG "vertically" (along the critical path(s)), making it difficult to formulate a cost function that can systematically take the *resource load* into consideration.

We found that the best results were obtained when operations were ordered lexicographically according to a three-component ranking function:

—*First Component*: the *alap*() value of the operation in the original DFG, with earlier operations considered first. Note that this component ensures that the producer of a value is bound before any of its consumers.
Consider, for example, the DAG in Figure 6. The partial order defined by *alap*() is:

$$\{v_1\} \to \{v_2, v_3, v_4\} \to \{v_5, v_6\}.$$

—*Second Component*: sorts the nodes at the same *alap*() level by their mobility $\mu(v)$, with lower mobility receiving higher priority.
For example, since $\mu(v_2) < \mu(v_3) = \mu(v_4)$ and $\mu(v_5) < \mu(v_6)$, the previous partial order is refined as

$$\{v_1\} \to \{v_2\} \to \{v_3, v_4\} \to \{v_5\} \to \{v_6\}.$$

—*Third Component*: orders the still unordered nodes by the number of consumers for an operation's result.
In our example, it resolves the order of v_3 and v_4:

$$\{v_1\} \to \{v_2\} \to \{v_3\} \to \{v_4\} \to \{v_5\} \to \{v_6\}.$$

Observe that the (partial) order defined by our ranking function still gives priority to operations on the critical path(s), so as to provide the most binding flexibility for those “time-sensitive” operations.

In addition, as shown in Section 3.1.2, the level-oriented priority function component enables the estimation of cluster load during the binding process, without the need to schedule the operations. This is one of the key features of our binding algorithm. It is very important because if fixed start times had been greedily assigned to operations during the binding process, it would have unnecessarily limited the flexibility of the remaining binding decisions.

3.1.2 *Cost Function.* Given an ordering on candidate nodes for binding, the next problem is to estimate the quality of various possible bindings for the node under consideration. Binding usually involves a trade-off between the delay associated with operation *serialization* (when an excessive load is placed on a cluster) and the delay due to insertion of data transfers (when the load is scattered through various clusters). For a cost function to work well, both of these delay “penalties” should be taken into account.

Accordingly, the cost $icost(v, c)$ of binding operation v to cluster c is expressed as

$$\begin{aligned} icost(v,c) = \; & fucost(v,c)\; dii(v)\,\alpha + buscost(v,c)\; dii(BUS)\,\beta \\ & + trcost(v,c)\; lat(BUS)\,\gamma, \end{aligned}$$

where $trcost(v,c)$ is the data transfer penalty, $fucost(v,c)$ is the FU serialization penalty, and $buscost(v,c)$ is the bus serialization penalty. As shown in the equation above, the penalties related to resource constraints, $fucost(v,c)$ and $buscost(v,c)$, are weighted by the data introduction interval $dii()$ of the corresponding resources. (If the resource for an operation v is not pipelined, $dii(v) = lat(v)$.) The penalty $trcost(v,c)$ associated with data transfer operations is weighted by bus latency $lat(BUS)$. We found that better results are obtained when the data transfer penalty is given just a slightly larger priority over the serialization penalties. This is achieved by the coefficients α, β, and γ (i.e., $\alpha = \beta = 1.0$ and $\gamma = 1.1$).

3.1.2.1 *Data Transfer Penalty* $trcost()$. The data transfer penalty function consists of two components:

$$trcost(v,c) = trcost_{dd}(v,c) + trcost_{cc}(v,c).$$

$trcost_{dd}$: Recall that our ordering of operations (Section 3.1.1) guarantees that, when we are binding an operation v, the producers of v's operands have already been bound. Thus, it is possible to calculate the number of data transfers required to deliver the operands to v, given a binding of v to a cluster c (see direct data dependency $(v1, v)$ in Figure 7). We denote this cost component as $trcost_{dd}(v,c)$, and call it the *direct data dependency component*. In order to calculate it, we consider all v's predecessors $pred(v)$ and, for each predecessor $u \in pred(v)$ bound to a different cluster $bn(u) \neq c$, we add 1 to the value of $trcost_{dd}(v,c)$. For example, in Figure 7, $pred(v) = v1$ and, since

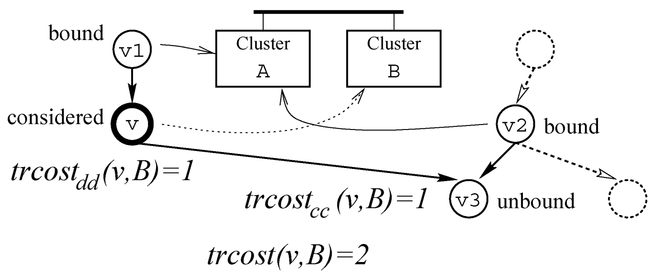


Fig. 7. Two components of the data transfer penalty function *trcost*().

$A = bn(v1) \neq B$, the direct data dependency component of *trcost*() for binding v to B is $trcost_{dd}(v, B) = 1$.

$trcost_{cc}$: When the operation being bound (see operation v in Figure 7) and a previously bound operation (v_2 in Figure 7) share a common consumer operation (v_3 in Figure 7), binding v to a cluster different from that assigned to v_3 will necessarily introduce a data transfer. We denote the cost component that captures such additional data transfer penalties by $trcost_{cc}(v, c)$, and call it the *common consumer component*. The common consumer component is calculated by considering all successors of v: we add 1 to the cost for each $u \in succ(v)$ that has a bound predecessor $w = pred(u)$, such that $bn(w) \neq c$. In Figure 7, $trcost_{cc}(v, B) = 1$ because $v3 \in succ(v)$ and $v2 = pred(v3)$, which is bound to cluster A: $bn(v2) = A \neq B$.

3.1.2.2 *FU Serialization Penalty fucost*(). To account for possible negative effects of serialization (delay in scheduling) of operations due to insufficient resources in a cluster, we consider an FU serialization penalty $fucost(v, c)$. We start by applying a relaxation technique similar to the one used in force-directed scheduling [Paulin and Knight 1987] to estimate the resource loads on a centralized datapath "equivalent."[7] When we consider the binding of operation $v \in V$ to a target cluster $c \in TS(v)$, we first calculate the corresponding resource load in c. Then the FU serialization penalty $fucost(v, c)$ of binding v to c is computed, by comparing the normalized load on c with the normalized load on the centralized datapath equivalent. Below we give a more detailed description of this process.

The resource load is expressed as a *load profile* over the "scheduling" steps, as shown in Figure 8. The load profile latency parameter L_{PR} is provided to the initial binding algorithm and may be varied, as described in Section 3.1.3.

[7]The decisions of force-directed scheduling are geared towards reducing operations concurrency (i.e., competition for resources) across all scheduling steps. In simple terms, when considering scheduling an operation on a given step, if the demand for resources at that step is above the average demand for the operation's range of alternative scheduling steps, then the force associated with that step is positive; otherwise it is negative. Thus, by selecting steps with minimum forces, the force-directed scheduling algorithm balances resource usage across the schedule. Our "load profile" method uses similar principles, but for the purpose of binding.

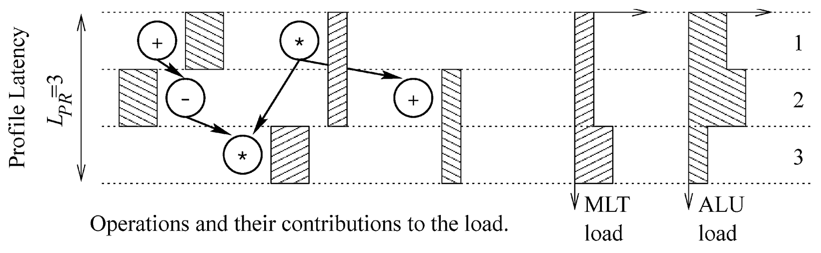


Fig. 8. Illustration of load profile.

Each operation v contributes to the load of the corresponding FU type t according to its time frame determined by L_{PR}. The load of operation v at level τ is defined as

$$load(v,\tau) = \begin{cases} 0 & \text{if } \tau < asap(v) \\ 0 & \text{if } \tau > alap(v) + dii(v) - 1 \\ \frac{1}{\mu_{pr}(v)+1} & \text{otherwise}, \end{cases}$$

where $\mu_{pr}(v) = alap(v) - asap(v)$ is the load profile mobility of v, with $alap()$ determined for a given L_{PR}. Note that the $asap()$, $alap()$, and the corresponding load profile are always calculated for the original DFG (i.e., without data transfers).

For every level τ and every FU type t, we define the normalized load profile of the centralized datapath $load_{DP}(t,\tau)$ as the sum of operation loads $load(v,\tau)$ at time τ for each operation v supported by FUs of type t. The load profile is normalized by $N(t)$, that is, by the number of FUs of type t in the datapath:

$$load_{DP}(t,\tau) = \sum_{v \in ops(t)} \frac{load(v,\tau)}{N(t)},$$

where $ops(t)$ is defined as $ops(t) = \{v \mid futype(optype(v)) = t\}$. Similarly, we define the normalized load profile on FUs of type t in cluster c as

$$load_{CL}(c,t,\tau) = \sum_{v \in ops(t),\ bn(v)=c} \frac{load(v,\tau)}{N(c,t)}.$$

As shown above, only bound operations ($bn(v) = c$) are considered in cluster load profiles.

In order to calculate $fucost(v,c)$, we temporarily update the load profile of the corresponding FU type t in cluster c and compare it with that of the centralized datapath equivalent. FU serialization penalty $fucost(v,c)$ is increased by 1 for each clock cycle τ for which $load_{CL}(c,t,\tau) > \max(load_{DP}(t,\tau), 1)$.[8] Note that the penalty is not incurred if $load_{CL}(c,t,\tau) \leq 1$, because it means that c is not overloaded, even if $load_{CL}(c,t,\tau) > load_{DP}(t,\tau)$.

3.1.2.3 *Bus Serialization Penalty buscost().* The efficiency and simplicity of the initial binding algorithm is partially based on the fact that we always work with the original DFG (i.e., our relaxation preserves the original level

[8]If the data introduction interval $dii(v) > 1$ (i.e., when not fully pipelined FUs are used), the load is extended beyond the operation's time frame. Thus, when comparing the load profiles, we may need to also look below the "current" level τ.

ordering of operations). A sufficiently good approximation of the bus load can be achieved by placing the data transfers "on the side," right after completion of the producing operation. The mobility of the data transfer is thus computed as the mobility of the corresponding consumer decreased by the bus latency $lat(BUS)$. If a data transfer tr "does not fit" (i.e., $alap(tr) > asap(tr)$), we assume $alap(tr)$ equal to $asap(tr)$, thus making $\mu(tr) = 0$. $buscost(v, c)$ is calculated by adding 1 for each clock cycle τ in which $load(BUS, \tau) > 1$. This approximation has worked well in practice, and is consistent with our use of the same centralized load profile (to calculate $fucost()$) throughout the entire binding process.

3.1.3 *Varying the L_{PR} Parameter.* The initial binding algorithm uses the load profile latency L_{PR} parameter (see Figure 8) for the purpose of calculating the load profiles of different resources. The top-level binding algorithm first sets L_{PR} equal to the critical path length L_{CP} of the original DFG. However, the actual best schedule length L^* achievable for a given datapath and DFG may be larger, due to unavoidable serializations and/or data transfers. If L_{CP} and L^* differ considerably, the estimations of resource load $fucost(v, c)$ and $buscost(v, c)$ may be overly pessimistic, which in turn may affect the quality of solutions produced by the initial binding algorithm B-INIT. Indeed, we found that increased profile latencies $L_{PR} > L_{CP}$ can frequently lead to better bindings in these cases.

Thus, a simple way to improve the binding quality is to run the initial binding algorithm for different profile latencies L_{PR} and choose the solution yielding the best estimated schedule latency. The top-level algorithm instructs the initial binding phase B-INIT as to how much L_{PR} should be "stretched." This approach is practical because of the low complexity of our initial binding.

3.1.4 *Reversing the Order of Binding.* We found that for some DFGs, especially the ones with a smaller number of inputs and larger number of outputs, starting the binding process from the output nodes may be beneficial. The initial binding algorithm remains essentially the same, with just a few symmetric changes. As in the previous case, this optimization is driven by the top-level binding algorithm.

3.2 Iterative Improvement Phase

Throughout our extensive experimental validation, the initial binding algorithm has performed very well, and in some cases we were able to verify that the generated solutions were identical in quality to those produced by the CPLEX ILP solver using the ILP formulation of simultaneous binding and scheduling for clustered datapaths proposed by Peixoto [1999]. However, in a significant number of cases, improvement was still possible. To take advantage of these opportunities, we developed an iterative improvement algorithm that uses specific binding optimizations aimed at "smoothing" the greediness of the initial binding, while still controlling computational complexity.

Specifically, our analysis has shown that the quality of the initial partitioning of nodes into clusters can be improved by focusing the optimizations on

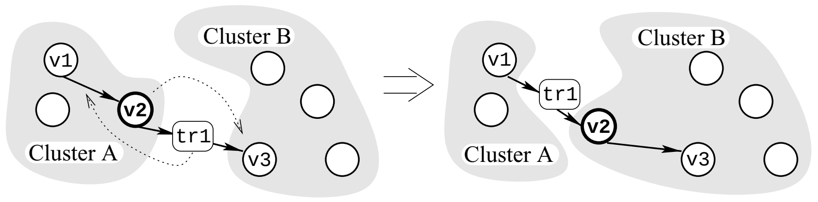


Fig. 9. Cluster boundary perturbation.

operations at the "boundaries" of the partitions defined by the current binding of operations to clusters, that is, on operations that have either producers or consumers bound to different clusters. For example, Figure 9 shows reassignment of one of such operations, $v2$, from cluster A to cluster B.

Iterative improvement based on such *boundary perturbations*, provides opportunities for repositioning, eliminating, and collapsing data transfers. Observe that in Figure 9, the data transfer operation $tr1$ "shifts" up along the path, possibly reducing bus congestion that may exist at the original temporal location of $tr1$. As far as regular operations are concerned, the perturbations can facilitate reduction of serialization in certain ways:

(1) by achieving a more favorable load distribution among clusters; and
(2) by shifting the scheduling positions of regular operations and data transfers up or down.

The latter is a result of modifications in the bound DFG (e.g., in Figure 9 the scheduling interval of $v2$ shifts down after cluster reassignment).

At each iteration in our improvement algorithm, we perform such boundary perturbations driven by a cost function. In its simpler version, the algorithm terminates when the perturbations fail to find a binding solution with cost improving upon that obtained in the previous iteration.[9] As illustrated in Figure 9, the boundary perturbations in each iteration are performed on the bound DFG, by considering all operations that have either an operand or result delivered to/from a different cluster. This is similar to the "neighborhood" concept in Geurts et al. [1997]. For each such operation, we temporarily rebind it to the cluster(s) where the operand/result resides. We perform such rebindings for individual operations and for pairs of operations. Each new binding produced by such perturbations is evaluated using a binding quality function.

3.2.1 *Binding Quality Functions.* In this section we discuss the two different quality functions developed for our algorithm: Q_U and Q_H. When the quality of the final binding solution is of major priority, we run two optimizations (one with Q_U and another with Q_H) and choose the result with the smaller estimated L (obtained with a fast list scheduler).

3.2.1.1 *Design Considerations.* For an iterative optimization process to work well, it is important for the quality function to facilitate a *gradual* (incremental) improvement from iteration to iteration.

[9]We discuss a more powerful option later in Section 3.2.1.

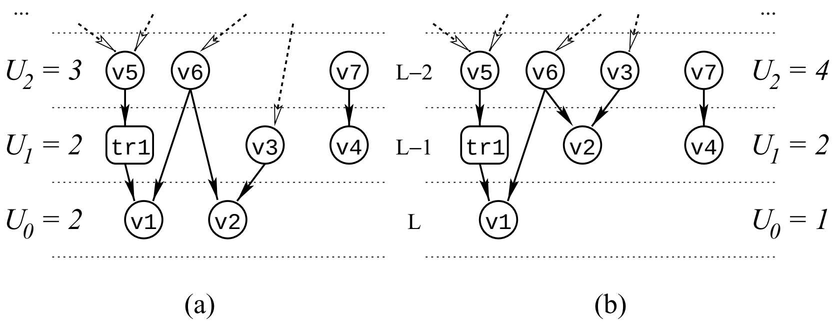


Fig. 10. Illustration of quality function Q_U.

Consider the schedule fragment shown in Figure 10(a), where two operations $v1$ and $v2$ are executed at the last clock cycle $\tau = L$. To improve the overall schedule latency L, both of these operations need to become schedulable at an earlier cycle, yet this may be impossible to achieve in a single perturbation iteration. Suppose, however, that one improvement iteration can find a binding that makes it possible to schedule operations $v3$ and $v2$ one clock cycle earlier without affecting $v1$ (see Figure 10(b)). Such modification does not change L. Thus, a naïve quality function that only considered the schedule latency would not distinguish between bindings (a) and (b) in Figure 10. Our experiments showed, however, that a binding like (b) very often has advantages over (a), since a single local perturbation iteration generally has more chances to improve the schedule latency L when fewer operations complete at the last clock cycle. This was especially noticeable in DFGs with a large number of outputs, such as the discrete cosine transform algorithms (see Section 4) or unrolled versions of single-output DFGs.

3.2.1.2 *The Q_U Function.* We developed a simple and very efficient quality function that is capable of estimating not only the quality of a binding, but also its potential for improvement of L. It is expressed as a vector $Q_U = (L, U_0, U_1, ...)$, where U_i is the number of regular operations completed at step $L - i$ (see Figure 10). Two bindings are compared lexicographically using the elements of their corresponding Q_U vectors.

3.2.1.3 *The Q_H Function.* The second binding cost function focuses on the overall delay incurred by regular operations due to serialization and/or data transfers. For each regular operation $v \in V$, we define an *operation delay* (see Figure 11) as follows.

$$\begin{aligned} H(v) &= \min_{v_i \in pred(v)} h_i, \quad \text{where} \\ h_i &= start(v) - finish(v_i) - 1 \\ &= start(v) - start(v_i) - lat(v_i). \end{aligned}$$

Note that $H(v)$ has two desirable properties. Its components h_i do not depend on the reason for the delay. Indeed, the definition of h_i does not specify what

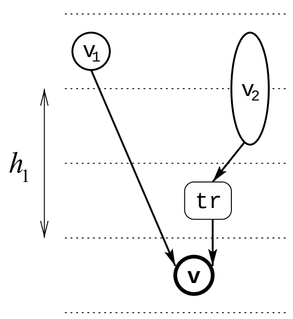


Fig. 11. Illustration of operation delay.

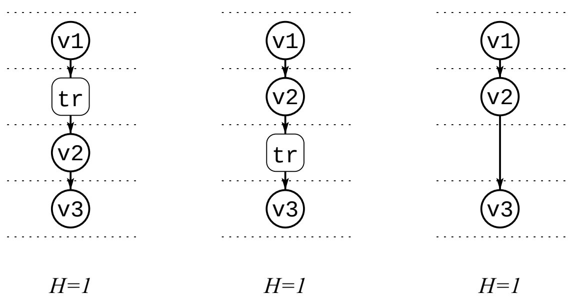


Fig. 12. Correct identification of equal solution quality.

caused the difference between *start*(v) and *finish*(v_i)—a data transfer between v_i and v or operation serialization due to cluster overload.

The second important property of $H(v)$ is that it only depends on the *smallest* component among h_i. Indeed, the fact that, in Figure 11, v is delayed by 2 cycles with respect to v_1 (i.e., $h_1 = 2$) should not and does not affect $H(v)$, because v can not be pushed up by more than *one* clock cycle ($h_2 = 1$), given the start time of v_2.

The total operation delay H is defined as follows.

$$H = \sum_{v \in V} H_\mu(v), \quad \text{where}$$

$$H_\mu(v) = \begin{cases} 0 & \text{if } \mu(v) \geq H(v) \\ H(v) - \mu(v) & \text{otherwise .} \end{cases}$$

In other words, H is the sum of operation delays in excess of their corresponding mobilities. The mobilities $\mu(v)$ here are calculated for the original (unbound) DFG with the target latency equal to the graph's critical path length: $L_{TG} = L_{CP}$. Observe that H provides a good integral characteristic of the solution quality and, unlike Q_U, pinpoints critical delays regardless of their proximity to the final clock cycle of the schedule. For example, all three solutions in Figure 12 have the same $H = 1$, and thus are correctly identified as equivalent.[10]

[10]For simplicity, we do not show other operation(s) that cause serialization of $v3$ in the right graph in Figure 12.

Table I. Benchmarks

Name	No. nodes	No. Connected Components	Critical Path
EWF	34	1	14
ARF	28	1	8
FFT	38	3	4
DCT-LEE	49	2	9
DCT-DIF	41	2	7
DCT-DIT	48	1	7
DCT-DIT-2	96	2	7
SWIM1	26	3	4
SWIM2	15	3	5
WUPWISE	14	2	3
QUAKE1	36	6	4
QUAKE2	36	6	4
AMMP	24	1	3

We use H to define our second binding quality function Q_H as follows.

$$Q_H = (L, H).$$

This quality function frequently works better than Q_U, especially in DFGs with few outputs.

3.2.1.4 *Minimizing Moves with Q_M.* As mentioned above, Q_U and Q_H are aimed at minimizing L. If additional minimization of N_{MV} is required, we use the quality function proposed by Desoli [1998] and denoted in this article by $Q_M = (L, N_{MV})$. Among solutions with the same estimated schedule latency L, this quality function gives preference to bindings with fewer data transfer operations.

4. EXPERIMENTAL RESULTS AND VALIDATION OF THE BINDING ALGORITHM

Table I summarizes key characteristics of the representative benchmarks selected for experimental validation of our algorithm. These include an elliptic wave filter (EWF), an autoregression filter (ARF), a version of a fast fourier transform (FFT) algorithm which is the main kernel in the RASTA benchmark from MediaBench [Lee et al. 1997], various discrete cosine transform (DCT) algorithms [Ifeachor and Jervis 1993], and the DCT-DIT-2, an unrolled version of the DCT-DIT algorithm. In order to assess the performance of the algorithms in a different application domain, we also selected some additional computationally intensive kernels from SPEC2000 (Floating Point) [Dixit 2001]: SWIM1, SWIM2, WUPWISE, QUAKE1, QUAKE2, and AMMP.

Throughout the examples in Tables II and III, we assume that the datapath has two buses and all operations take one cycle. In the last set of examples (Table IV), we vary the latency of data transfer operations and the number of buses for the FFT benchmark and also report the results for DCT-DIT with two-cycle multiplications, so as to illustrate the generality of the algorithm.

For each benchmark, several experiments were created using a broad variety of datapath configurations. Clusters are symbolically represented as $|i, j|$, where i is the number of ALUs, and j is the number of multipliers in the

Table II. Multimedia Benchmark Results for $N_B = 2$ and $lat(BUS) = 1$

Datapath	**PCC**		**B-INIT**			**B-ITER**		
	L/M	msec	L/M	ΔL%	msec	L/M	ΔL%	sec
DCT–DIF								
|1, 1|1, 1|	16/15	3.7	15/2	6.7	2.4	15/2	**6.7**	0.05
|2, 1|2, 1|	11/0	4.8	11/10	0	2.4	10/6	**10**	1.3
|2, 1|1, 1|	11/12	5.9	11/6	0	2.4	10/6	**10**	0.19
|1, 1|1, 1|1, 1|	12/8	13	12/9	0	3.1	11/8	**9**	5.1
DCT–LEE								
|1, 1|1, 1|	16/11	8.0	16/7	0	4.3	16/6	**0**	3.8
|2, 1|2, 1|	12/8	9.2	12/2	0	4.3	12/2	**0**	2.9
|2, 1|1, 1|	13/9	13	13/5	0	4.3	13/3	**0**	0.52
|2, 2|2, 1|	11/0	8.4	10/2	10	4.3	10/1	**10**	0.03
|1, 1|1, 1|1, 1|	14/8	19	12/14	17	5.5	12/10	**17**	3.7
DCT–DIT								
|1, 1|1, 1|	19/18	8.1	19/7	0	2.9	19/7	**0**	0.85
|2, 1|2, 1|	13/18	7.1	13/7	0	2.9	12/7	**8.3**	1.3
|1, 1|1, 1|1, 1|	15/18	7.3	15/19	0	3.7	13/15	**15**	7.3
|2, 1|2, 1|1, 1|	12/6	11	11/13	9	3.7	11/9	**9**	1.5
|3, 1|2, 2|1, 3|	11/12	15	11/12	0	3.7	9/9	**22**	3.1
|1, 1|1, 1|1, 1|1, 1|	14/17	22	13/17	7.7	4.4	11/14	**27**	7.4
DCT–DIT–2								
|1, 1|1, 1|	37/32	20	37/14	0	5.8	37/13	**0**	2.2
|2, 1|2, 1|	23/28	38	23/17	0	5.8	22/23	**4.6**	20
|1, 1|1, 1|1, 1|	25/28	29	27/15	−7.4	7.3	25/13	**0**	16
|3, 1|2, 2|1, 3|	17/18	43	17/20	0	8.2	14/20	**21**	22
|1, 1|1, 1|1, 1|1, 1|	22/30	174	20/21	10	9.0	19/18	**16**	21
FFT								
|1, 1|1, 1|	14/6	5.8	14/4	0	1.9	14/4	**0**	0.10
|2, 1|2, 1|	10/6	7.7	10/4	0	1.9	10/4	**0**	0.14
|1, 1|1, 1|1, 1|	12/8	6.1	10/12	20	2.4	10/9	**20**	1.5
|2, 1|2, 1|1, 2|	10/4	9.8	8/10	25	2.6	8/5	**25**	0.6
|3, 2|3, 1|1, 3|	7/4	13	7/6	0	2.6	6/5	**17**	1.8
|1, 1|1, 1|1, 1|1, 1|	11/10	25	10/12	10	3.0	9/6	**22**	5.4
EWF								
|1, 1|1, 1|	18/5	5.7	17/3	5.9	3.9	17/3	**5.9**	0.04
|2, 1|2, 1|	15/2	4.1	16/3	−6.3	3.9	15/1	**0**	1.5
|2, 1|1, 1|	15/2	4.2	16/5	−6.3	3.9	15/3	**0**	0.59
|1, 1|1, 1|1, 1|	18/5	18	17/7	5.9	4.8	16/5	**12**	1.4
|2, 2|2, 1|1, 1|	15/2	7.2	15/5	0	4.9	14/5	**7.1**	3.3
ARF								
|1, 1|1, 1|	13/5	1.6	11/4	18	2.0	11/4	**18**	0.22
|1, 2|1, 2|	10/5	2.0	10/5	0	2.0	10/4	**0**	0.28

corresponding cluster.[11] For comparison purposes, we also report schedule latencies and number of data transfer operations obtained with our implementation of the Partial Component Clustering (PCC) algorithm [Desoli 1998; Faraboschi et al. 1998], one of the best binding algorithms found in the literature (see Section 5).

[11]For example, the configuration |2, 1|1, 1| represents a datapath with two clusters. The first one has two ALUs and one multiplier; the second one includes one ALU and one multiplier.

Table III. SPEC2000 Benchmark Results for $N_B = 2$ and $lat(BUS) = 1$

Datapath	**PCC**		**B-INIT**			**B-ITER**		
	L/M	msec	L/M	ΔL%	msec	L/M	ΔL%	sec
SWIM1								
|1, 1|1, 1|	9/1	1.5	9/1	0	0.63	9/1	**0**	0.03
|2, 1|2, 1|	7/1	1.4	7/1	0	0.63	7/1	**0**	0.03
|2, 1|1, 1|	8/0	1.9	7/2	14	0.63	7/2	**14**	0.02
|2, 2|2, 1|	6/0	1.8	6/1	0	0.63	6/0	**0**	0.02
|1, 1|1, 1|1, 1|	7/5	2.4	6/3	17	0.77	6/3	**17**	0.03
|1, 1|1, 1|1, 1|1, 1|	7/0	4.5	6/4	17	0.99	6/2	**17**	0.12
SWIM2								
|1, 1|1, 1|	6/1	0.82	7/0	−14	0.47	6/1	**0**	0.01
|2, 1|2, 1|	6/0	0.59	6/0	0	0.47	6/6	**0**	0.01
|2, 1|1, 1|	6/0	0.61	6/0	0	0.47	6/0	**0**	0.01
|2, 2|2, 1|	5/0	0.58	5/0	0	0.47	5/0	**0**	0.01
|1, 1|1, 1|1, 1|	5/0	2.7	5/0	0	0.59	5/0	**0**	0.01
|1, 1|1, 1|1, 1|1, 1|	5/0	4.2	5/5	0	0.7	5/0	**0**	0.01
WUPWISE								
|1, 1|1, 1|	6/4	0.62	6/2	0	0.24	6/0	**0**	0.01
|2, 1|2, 1|	6/4	0.62	7/2	−14	0.24	6/2	**0**	0.02
|2, 1|1, 1|	6/4	0.62	7/2	−14	0.24	6/2	**0**	0.03
|2, 2|2, 1|	5/4	0.98	5/1	0	0.24	5/1	**0**	0.02
|1, 1|1, 1|1, 1|	6/2	1.8	6/0	0	0.23	5/2	**20**	0.03
|1, 1|1, 1|1, 1|1, 1|	5/2	2.1	5/2	0	0.37	5/2	**0**	0.02
QUAKE1								
|1, 1|1, 1|	10/0	1.6	10/0	0	0.86	10/0	**0**	0.09
|2, 1|2, 1|	10/0	1.7	10/0	0	0.86	10/0	**0**	0.17
|2, 1|1, 1|	10/0	1.7	10/3	0	0.86	10/2	**0**	0.17
|2, 2|2, 1|	7/0	1.6	7/0	0	0.86	7/0	**0**	0.08
|1, 1|1, 1|1, 1|	7/0	2.7	7/0	0	1.2	7/0	**0**	0.09
|1, 1|1, 1|1, 1|1, 1|	7/0	3.2	7/3	0	1.4	6/2	**17**	0.43
QUAKE2								
|1, 1|1, 1|	13/0	1.8	13/0	0	0.85	13/0	**0**	0.03
|2, 1|2, 1|	11/0	1.8	11/0	0	0.85	11/0	**0**	0.03
|2, 1|1, 1|	13/0	1.8	12/3	8.3	0.85	12/1	**8.3**	0.06
|2, 2|2, 1|	9/0	2.6	8/3	13	0.85	8/3	**13**	0.03
|1, 1|1, 1|1, 1|	9/0	3.0	9/0	0	1.1	9/0	**0**	0.09
|1, 1|1, 1|1, 1|1, 1|	9/0	3.4	9/0	0	1.4	9/0	**0**	0.11
AMMP								
|1, 1|1, 1|	8/5	2.8	8/5	0	0.43	8/5	**0**	0.09
|2, 1|2, 1|	7/4	2.7	8/5	−13	0.43	7/4	**0**	0.10
|2, 1|1, 1|	7/4	2.7	8/8	−13	0.43	7/4	**0**	0.18
|2, 2|2, 1|	6/5	2.7	6/5	0	0.43	6/3	**0**	0.14
|1, 1|1, 1|1, 1|	7/6	5.0	7/6	0	0.55	6/6	**17**	0.32
|1, 1|1, 1|1, 1|1, 1|	8/4	6.6	7/9	14	0.63	6/7	**33**	1.3

Tables II through IV present "schedule latency/number of data transfers" pairs (L/M) for the PCC algorithm, for our initial binding phase (B-INIT), and for our iterative improvement phase (B-ITER). They also show the latency improvement percentages ($\Delta L\%$) of our algorithm as compared to PCC and the CPU times. The performance of the algorithms is also summarized in Figure 13. Our B-INIT algorithm almost always executes faster than PCC (which includes

Table IV. Example of Binding Results for FFT and DCT-DIT with Several Values of N_B, $lat(BUS)$, and $lat(MLT)$

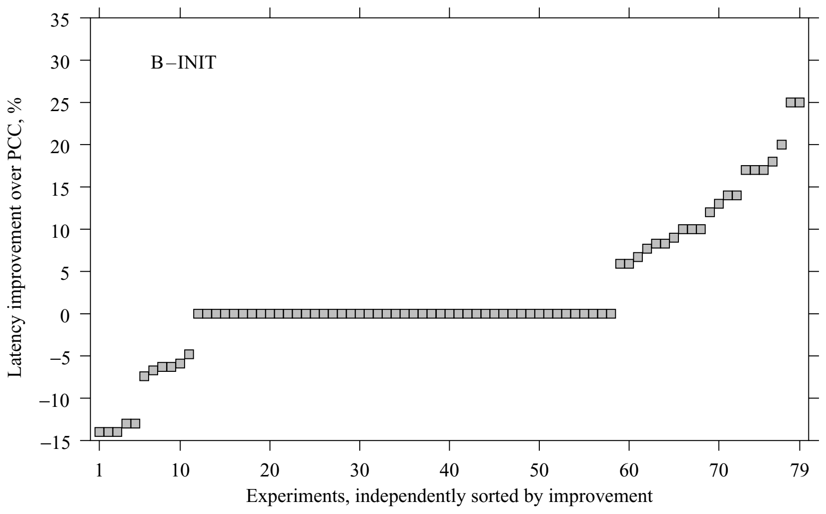
Datapath		**PCC**		**B-INIT**			**B-ITER**		
Clusters	N_B	L/M	msec	L/M	ΔL%	msec	L/M	ΔL%	sec
FFT — $lat(BUS) = 1,\ lat(MLT) = 1$									
|2, 2|2, 1|2, 2|3, 1|1, 1|	1	9/5	19	8/4	12	1.8	7/4	**29**	5.0
	2	8/4	35	8/4	0	1.8	7/5	**14**	1.9
FFT — $lat(BUS) = 2,\ lat(MLT) = 1$									
|2, 2|2, 1|2, 2|3, 1|1, 1|	1	10/5	21	8/4	25	1.8	8/2	**25**	5.7
	2	8/4	28	8/4	0	1.8	7/4	**14**	7.4
DCT–DIT — $lat(BUS) = 1, lat(MLT) = 2$									
|1, 1|1, 1|	2	20/18	8.5	21/15	−4.8	4.3	20/7	**0**	0.76
|2, 1|2, 1|	2	14/14	7.4	15/6	−6.7	4.2	13/7	**7.7**	1.2
|1, 1|1, 1|1, 1|	2	16/18	7.7	17/19	−5.9	5.5	14/14	**14**	8.0
|2, 1|2, 1|1, 1|	2	13/16	12	13/17	0	5.4	12/13	**8.3**	2.6
|3, 1|2, 2|1, 3|	2	13/14	12	12/10	8.3	5.6	11/7	**18**	2.9
|1, 1|1, 1|1, 1|1, 1|	2	15/17	24	15/18	0	6.7	13/13	**15**	9.1

an iterative improvement phase responsible in some cases for as much as a 1900% slowdown as compared to B-INIT). Yet, in the majority of the examples (see Figure 13), B-INIT performs no worse than PCC, and even shows latency improvements of up to 20 to 25% in some cases. B-ITER, our second binding phase shows more consistent improvements over PCC, at the expense of an increase of computation time (up to 7 seconds in some datapath configurations for DCT-DIT and FFT and to 22 seconds for 96-node DCT-DIT-2, measured on an RS6000 595). Note that the iterative improvement binding algorithm aims at high optimization. We consider these times acceptable in the context of the embedded applications of interest, since the quality of the synthesized VLIW datapaths and/or the quality of generated code are of major importance.

We have also conducted an additional set of experiments to assess the robustness of the binding algorithm over a wide range of datapath configurations, particularly for datapaths with an "excessive" number of clusters.[12] It has been confirmed that the proposed binding algorithm successfully resists unnecessary scattering of operations which is frequently a problem with other binding algorithms if they are solely based on load balancing (some such results are shown in Table IV).

A final note concerning the experimental results presented in this section: the schedule latencies reported in Tables II through IV were derived by a list scheduler, using the specific binding (cluster assignment) function generated by each of the algorithms PCC, B-INIT, and B-ITER. Register files are assumed to have unlimited size, and thus register allocation is not included in the scheduling process. All other resources, namely, functional units, clusters, and buses, as well as operations (including move/copy), are accurately modeled by the scheduler, as discussed in Section 2.

[12]By an "excessive" number we mean the case when the DFG is such that any binding which utilizes all the clusters leads to a worse schedule than a good binding that leaves some of the clusters completely unassigned.



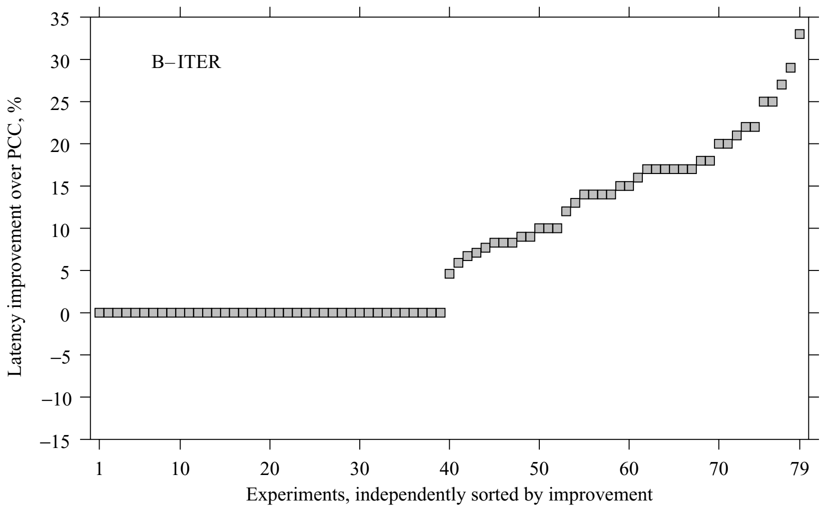


Fig. 13. Summary of relative performance of B-INIT and B-ITER compared to PCC. Each point represents a single experiment.

PCC and B-ITER use actual schedule latencies to evaluate the quality of candidate solutions, and direct their iterative search. Those latencies are generated by an external scheduler. In contrast, B-INIT, our constructive greedy algorithm, does not include actual schedule latencies in its cost function (i.e., works based solely on abstract metrics; see Section 3.1). Accordingly, since it is conceivable that some of the latency results reported in Tables II through IV may not be possible to achieve on a machine with limited register file sizes (due to spills), there is no guarantee that B-INIT would indeed perform better than (or similarly to) PCC or B-ITER in such cases. Still, the specific design of B-INIT's cost function (see Section 3.1.2), explicitly incorporating serialization (i.e., cluster oversubscription) and data transfer penalties, has proven to be very powerful. Indeed, as discussed in the sequel, we have been able to easily adapt it so as to generate binding solutions that lead to different degrees of register pressure during scheduling.

Specifically, throughout the experiments performed with the register-sensitive software pipelining algorithm CALiBeR [Akturan and Jacome 2002], we empirically found that decreases in the relative weight of the data transfer penalty coefficient γ with respect to the cluster serialization penalty coefficient α in B-INIT's cost function, lead to finding binding solutions with "inherently" lower register pressure. Indeed, by doing so, one favors solutions that more aggressively distribute load across clusters, thus decreasing ILP on each individual cluster. For details on the excellent performance of the software pipelining algorithm CALiBeR versus state-of-the-art approaches, see Akturan and Jacome [2002].

We conclude by observing that the comparative assessment of the performance of B-ITER versus PCC, based on the empirical data shown in Tables II through IV, is not significantly affected by the fact that register allocation is not incorporated in the external scheduler used to produce the latency results. Indeed, both iterative improvement algorithms account for the specific idiosyncrasies of the scheduler being used (and, for that matter, of a possible register allocator as well) while generating their solutions, since both incorporate the actual schedule latency of candidate solutions (produced by the external scheduler), in their cost functions. Thus, if register allocation is incorporated, both B-ITER and PCC will take its impact into consideration. Still, when targeting machines with "small" register files, it may be of interest to run B-ITER for a few initial solutions, produced by varying the relative weights of the parameters α and γ in B-INIT's cost function, as mentioned above.

5. PREVIOUS WORK

Capitanio et al. [1992] perform binding using an extension of classical network partitioning algorithms with simulated annealing enhancements. The algorithm is given the initial schedule obtained for an equivalent centralized machine. The primary cost function is the size of the cutset. The underlying idea is that limiting the communication (number of moves) between clusters minimizes the increase in the schedule length due to clustering. Unfortunately, the load balancing among clusters induced by the algorithm is not a guarantee of latency minimization. In fact, in our experiments we found that sometimes the highest quality solution executes only a small subset of the operations in some of the clusters. Moreover, due to the specifics of the algorithm in Capitanio et al. [1992], the target architecture must have homogeneous clusters; that is, all clusters must have exactly the same number and type of FUs. Similarly to ours, the algorithm was tested on a number of basic block kernels.

Leupers [2000] presents an "instruction partitioning" (binding) algorithm where an initial (random) binding is improved by simulated annealing. A detailed scheduling is performed for each generated binding and the corresponding latency is used as a cost function driving the optimization. The author reports that the algorithm is not sensitive to the quality of the initial binding and thus a random initial binding is performed. The cost function evaluated at each simulated annealing iteration is the schedule latency obtained by a detailed scheduling algorithm. (The approach was experimentally validated for the two-cluster Texas Instruments' C6201 VLIW processor.) The experiments show from 7 to 26% improvement in schedule latency, as compared to the TI assembly optimizer, at the expense of an increase in compilation time typical of simulated annealing algorithms. Unfortunately, the execution time of the algorithm is likely to be significantly affected if one considers machines with a larger number of clusters. Our improvement algorithm is less sensitive to the number of clusters since only boundary perturbations are performed (see Section 3.2), starting from a "good" initial point. Similarly to our experiments, the author uses time-critical basic blocks from typical DSP algorithms, without considering register allocation.

Hanno and Devadas [1998] address several aspects of code generation including binding for a clustered datapath architecture. They introduce an elegant model, the *split-node direct acyclic graph*, which captures all possible operation bindings along with the necessary data transfers. However, the binding algorithm based on that model only handles clusters with a single functional unit.

Özer et al. [1998] present a greedy binding/scheduling algorithm similar to our initial binding algorithm. In contrast to our cost function (Section 3.1.2), theirs requires the computation of ready times for operations being bound (and thus, their scheduling). Although the algorithm in Ožer et al. [1998] also outputs a schedule, we believe that a postprocessing scheduling step would still be beneficial, since the scheduler would then have complete information on all the modifications in the DFG induced by the binding. The authors use inner loop basic blocks (selected from benchmark programs) to evaluate the algorithm.

Several research groups (see, e.g., Nystrom and Eichenberger [1998]; Fernandes et al. [1999], and Sanchez and Gonzàlez [2001]) address binding in the context of *modulo scheduling* algorithms. The objective of modulo scheduling is to software pipeline the inner loop body (i.e., derive a retiming function for its operations), as well as determine adequate binding and scheduling functions, so as to minimize the loop's initiation interval (i.e., maximize throughput). The problem addressed in those papers is different from the problem addressed in this work, in that they are performing performance-enhancing loop transformations. In fact, as mentioned earlier, our initial binding algorithm B-INIT was used as a "tool" inside the optimization process of CALiBeR, a software pipelining algorithm proposed by Akturan and Jacome [2001]. A trivial extension was needed to incorporate recurrence edges in the algorithm (see Akturan and Jacome [2001]).

Kailas et al. [2001] describe a code generation framework for clustered VLIW processors. This work is similar to Özer et al. [1998] in that it combines binding and scheduling. In addition, it includes a new on-the-fly global and local register allocation method. Specifically, the algorithm uses a greedy strategy based on list scheduling to perform simultaneous binding, scheduling, and register allocation. Upon selecting an operation from the ready list (priority is given to operations currently on the critical path), the schedule cycle[13] of that operation is calculated for each of the clusters and the operation is bound to the cluster with the earliest cycle (tie-breaking is based on the presence of extra copy operation(s) or on register pressure). The scheduling and register allocation (and if necessary, copy and spill code insertion) is then performed for that operation.[14]

Desoli [1998] and Faraboschi et al. [1998] developed an elegant two-phase binding algorithm called partial component clustering.[15] The first phase of PCC partitions the DFG into several partial components, using a depth-first traversal, similarly to the bottom-up greedy (BUG) [Ellis 1986] algorithm. Several such partitions are created by varying the threshold parameter, the maximum number of nodes per partial component. An initial assignment algorithm then places the partial components into clusters, trying to balance the load and minimize intercluster communication. The second phase implements an iterative improvement of the initial binding, driven by the Q_M cost function (see Section 3.2) with latency obtained by a fast scheduler which includes an estimator for register allocation. We found this algorithm to be one of the best representatives of the state of the art, and thus selected it to be our reference algorithm (see Section 4).

Mattson et al. [2000] argue the advantages of distributed register file VLIW architectures over the clustered architectures targeted in this article. In a distributed register file architecture, each functional unit is connected to the single read port of a dedicated register file, and all functional unit outputs are connected by shared buses to the single shared write port of each register file [Mattson et al. 2000]. Note that scheduling in the context of such architectures is significantly more complex than scheduling for clustered architectures, since it requires simultaneous: (1) allocation/binding of operations to individual functional units; (2) allocation/binding of buses and register file ports to the communications between such operations (route assignment); and (3) scheduling of operations and routes (i.e., communications between operations). An elegant, incremental scheduling algorithm is proposed that interleaves allocation and scheduling of operations with allocation and scheduling of communications, carefully searching over permutations of alternative read (or write) communication paths, and inserting copy operations as needed. As with our (binding only) algorithm, their proposed communications scheduling algorithm does not consider register allocation, that is, essentially assume that the register files instantiated in the machine have infinite size, and rely on a postpass register allocation, spill insertion, and scheduling step to account for their limited size. Note that this assumption may be more problematic in the context of distributed register file architectures than in the

[13]This evaluation takes into account data dependency and resource constraints, including possible copy operations and the availability of dead registers.
[14]More precisely, this is only performed if the operation's earliest cycle is on or before the current cycle. Otherwise, the current operation is dropped and the next one is considered.
[15]Faraboschi et al. [1998] present a good overview of clustering in general for VLIW datapaths.

context of clustered architectures, since the numerous individual register files in the former (one per functional unit input port), are likely to be smaller than their counterparts in a clustered architecture, and thus more vulnerable to overflow due to "local spikes" in the number of live variables. This is, however, the price of keeping the algorithm complexity under control. Unfortunately, it is not clear that the proposed algorithm will scale for the large number of functional units envisioned by the authors for such machines: on the order of hundreds. [Mattson et al. 2000] Indeed, because of the significant complexity of distributed register file architectures, in order to ensure high quality results, the algorithm performs a search over permutations of valid read (or write) communication paths, which is exponential with the number of communications, that is, with ILP [Mattson et al. 2000]. Thus the exponential growth in execution time of the algorithm may preclude its utilization in the very high ILP context for which such machines are primarily designed.

A final note concerning the empirical contrast between clustered and distributed register file organizations presented in the article: specifically, for a set of representative kernels, the authors report that a distributed register file architecture with 12 functional units delivers significant gains in performance (up to 120%), as compared to: (1) a clustered architecture with 2 clusters, each with 6 functional units; and (2) a clustered architecture with 4 clusters, each with 3 functional units. Inspecting the detailed data provided in the article for the test kernels, we found it surprising that the two comparison clustered architectures (with such different cluster granularities), delivered exactly the same average performance over the set of experiments; that is, both delivered an average speedup of 0.82 in schedule length, as compared to an "ideal" centralized machine with the same number of functional units. Moreover, the machine with the smaller clusters frequently outperformed the machine with the larger clusters in the experiments. These results are in sharp contrast with the empirical results reported, for example, by Faraboschi et al. [1998] and Lapinski et al. [2002], where it was found that increasing cluster size (i.e., the number of functional units in a cluster) consistently improved schedule latency. Accordingly, it would be interesting to see how these comparative performance numbers might be affected if a high-quality binding algorithm, such as the one proposed by Desoli [1998], or in this article, were used for the experiments with the clustered machines.

6. CONCLUSIONS

We proposed an effective binding algorithm for clustered VLIW processors and experimentally demonstrated performance improvements of up to 33% as compared to one of the best state-of-the art binding algorithms reported in the literature. Beyond its relevance to code generation, due to its flexibility and efficiency, our binding algorithm has been successfully incorporated in a register-sensitive software pipelining algorithm for clustered VLIW machines [Akturan and Jacome 2001] and in a design space exploration framework for datapath configurations of application-specific VLIW processors [Lapinskii et al. 2002].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

AKTURAN, C. AND JACOME, M. F. 2001. CALiBeR: A software pipelining algorithm for clustered VLIW processors. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*. 112–118.

AKTURAN, C. AND JACOME, M. F. 2002. An effective software pipelining algorithm for clustered embedded VLIW processors. *J. Des. Autom. Embed. Sys.*, Special Issue on Design Methodologies and Tools for Real-Time Embedded Systems (to appear).

ANALOG DEVICES. 2001. ADSP-TS001M TigerSHARC DSP product description. Available online at http://www.analog.com/products/descriptions/ADSP-TS001.html.

BASOGLU, C., ZHAO, K., KOJIMA, K., AND KAWAGUCHI, A. 2000. The MAP-CA VLIW-based media processor. Equator Technologies Inc. and Hitachi Ltd. Available online at http://equator.com.

CAPITANIO, A., DUTT, N., AND NICOLAU, A. 1992. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture* (Portland, OR.), 292–300.

COLWELL, R., HALL, W., JOSHI, C., PAPWORTH, D., RODMAN, P., AND TORNES, J. 1990. Architecture and implementation of a VLIW supercomputer. In *Proceedings of Supercomputing '90*. (Branford, CT), 910–919.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. Mc Graw-Hill, New York.

DESOLI, G. 1998. Instruction assignment for clustered VLIW DSP compilers: A new approach. Tech. Rep. HPL-98-13, Hewlett-Packard Co., February.

DIXIT, K. 2001. Performance SPECulations—Benchmarks, friend or foe. In *Procedings of the Seventh International Symposium on High Performance Computer Architecture* (Monterrey, Mexico).

EBCIOĞLU, K., FRITTS, J., KOSONOCKY, S., GSCHWIND, M., ALTMAN, E., KAILAS, K., AND BRIGHT, T. 1998. An eight-issue tree-VLIW processor for dynamic binary translation. In *Proceedings of the International Conference on Computer Design (ICCD'98)* (IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.), Piscataway, N.J., IEEE Press, 488–495.

ELLIS, J. R. 1986. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, Mass.

FARABOSCHI, P., BROWN, G., FISHER, J. A., AND DESOLI, G. 2000. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, B.C.).

FARABOSCHI, P., DESOLI, G., AND FISHER, J. A. 1998. Clustered instruction-level parallel processors. Tech. Rep. HPL-98-204, Hewlett-Packard Co., December.

FERNANDES, M. M., LLOSA, J., AND TOPHAM, N. 1999. Distributed modulo scheduling. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture* (Dept. of Comput. Sci., Edinburgh Univ., UK), 130–134.

FRITTS, J., WU, Z., AND WOLF, W. 1999. Parallel media processors for the billion-transistor era. In *Proceedings of the International Conference on Parallel Processing* (Aizu, Japan).

GEURTS, W., CATTHOR, F., VERNALDE, S., AND DEMAN, H. 1997. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic, Hingham, Mass.

HANNO, S. AND DEVADAS, S. 1998. Instruction selection, resource allocation and scheduling in the AVIV retargetable code generator. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, 510–515.

IFEACHOR, E. AND JERVIS, B. 1993. *Digital Signal Processing: A Practical Approach*. Addison-Wesley, New York.

JACOME, M. F. AND DE VECIANA, G. 2000. Design challenges for new application specific processors. *IEEE Des. Test Comput. 17*, 2 (April–June), 40–50.

KAILAS, K., EBCIOĞLU, K., AND AGRAWALA, A. 2001. CARS: A new code generation framework for clustered ILP processors. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture* (Monterrey, Mexico).

LAPINSKII, V., JACOME, M. F., AND DE VECIANA, G. 2002. Application-specific clustered VLIW datapaths: Early exploration on a parameterized design space. *IEEE Trans. Comput. Aid. Des. Integ. Circ. Syst.* (accepted for publication).

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 330–335.

LEUPERS, R. 2000. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques* (Philadelphia).

MATTSON, P., DALLY, W. J., RIXNER, S. W., KAPASI, U. J., AND OWENS, J. D. 2000. Communication scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 82–92.

NYSTROM, E. AND EICHENBERGER, A. E. 1998. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st Annual International Symposium on Microarchitecture* (Dallas), 3–13.

ÖZER, E., BANERJIA, S., AND CONTE, T. 1998. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*.

PAULIN, P. G. AND KNIGHT, J. P. 1987. Force-directed scheduling in automatic data path synthesis. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, Miami Beach, 195–202.

PEIXOTO, H. P. 1999. Reuse and estimation techniques for embedded systems-on-a-chip. PhD Thesis, The University of Texas at Austin.

RAU, B. R., KATHAIL, V., AND ADITYA, S. 1998. Machine-description driven compilers for EPIC processors. Tech. Rep. HPL-98-40, Hewlett-Packard Co., September.

RIXNER, S., DALLY, W. J., KHAILANY, B., MATTSON, P., KAPASI, U. J., AND OWENS, J. D. 1999. Register organization for media processing. In *Proceedings of the 26th International Symposium on High-Performance Computer Architecture*.

SÁNCHEZ, J. AND GONZÁLEZ, A. 2000. Instruction scheduling for clustered VLIW architectures. In *Proceedings of the thirteenth International Symposium on System Systhesis (ISSS-13)*. (Madrid).

TEXAS INSTRUMENTS. 2000. TMS320C6000 CPU and instruction set reference guide. Literature Number: SPRU226.

WHITE, S. W. AND DHAWAN, S. 1994. POWER2: Next generation of the RISC System/6000 family. *IBM J. Res. Dev. 38*, 5 (Sept.), 493–502.

Received June 2001; revised April 2002; accepted April 2002