MPI microtask for programming the Cell Broadband Engine™ processor

M. Ohara
H. Inoue
Y. Sohda
H. Komatsu
T. Nakatani

The Cell Broadband Engine™ processor employs multiple accelerators, called synergistic processing elements (SPEs), for high performance. Each SPE has a high-speed local store attached to the main memory through direct memory access (DMA), but a drawback of this design is that the local store is not large enough for the entire application code or data. It must be decomposed into pieces small enough to fit into local memory, and they must be replaced through the DMA without losing the performance gain of multiple SPEs. We propose a new programming model, *MPI microtask*, based on the standard Message Passing Interface (MPI) programming model for distributed-memory parallel machines. In our new model, programmers do not need to manage the local store as long as they partition their application into a collection of small microtasks that fit into the local store. Furthermore, the preprocessor and runtime in our microtask system optimize the execution of microtasks by exploiting explicit communications in the MPI model. We have created a prototype that includes a novel static scheduler for such optimizations. Our initial experiments have shown some encouraging results.

INTRODUCTION

The Cell Broadband Engine** (BE) processor[1] is an asymmetric multicore processor that combines a general-purpose IBM PowerPC* processor element (PPE) and eight synergistic processor elements (SPEs).[2] From an architectural standpoint, this processor has a high peak performance because the SPE is simpler and more efficient than general-purpose processors in terms of the micro and memory architecture.[3] One architectural aspect is the small high-speed local store at each SPE. Because the size of the local store is limited to a range of L2-cache sizes—256 KB for the first-generation Cell BE processor—many real-world applications do not fit in the local store. While conventional microprocessors have a hardware cache to manage such a small local store, the Cell BE processor must rely on a software mechanism to manage it. This requirement for software manage-

ment could impose significant challenges to programmers, but at the same time it offers significant opportunities for the software to take advantage of the raw performance of the Cell BE processor.

The microtask we propose here provides a programming model that frees programmers from local-store management and enables the preprocessor and runtime system to optimize the scheduling of computations and communications by taking advantage of the explicit communication model in the Message Passing Interface (MPI).[4,5] In the microtask model, programmers are still responsible for partitioning the application into multiple microtasks. Each microtask is essentially a virtualized SPE that uses the MPI to communicate with other microtasks.

We have chosen MPI as a communication application programming interface (API) for the following two reasons. First, the Cell BE processor adopts a distributed-memory model; the PPE and SPEs use direct memory access (DMA) operations for communications. Thus, the overhead due to a message-passing layer can be inherently small because of the commonality between the native hardware and the message-passing model. The model, moreover, can hide hardware details from programmers. Second, and perhaps more important, the message-passing model allows us to analyze the dependency between microtasks by examining message APIs. Such dependency information is essential for various optimizations in task and communication management. Among existing message-passing interfaces, we selected MPI because it is widely used as a standard interface.

Our microtask system provides a *preprocessor* that transforms a microtask program in the message-passing model to one in a streaming model[2] that the Cell BE processor can execute efficiently. To do this, the preprocessor first divides each microtask into a collection of *basic tasks*, each of which represents a unit of computation that causes communication only at its beginning and end. Thus, each basic task corresponds to a computation kernel in stream programming languages[6,7] in the sense that the concept of the basic task separates computation from communication. This separation allows the preprocessor to schedule basic tasks in such a way that data streams through SPEs over high-speed, on-chip DMA channels.

To make the streaming model effective, the preprocessor then puts basic tasks with strong dependencies together as a cluster and applies a heuristic algorithm to schedule clusters. The cluster-scheduling algorithm creates a precedence graph of clusters in a series-parallel form[8] and then applies a dynamic programming algorithm. The nested structure of the series-parallel graph allows the dynamic programming algorithm to reuse partially scheduled results to reduce scheduling time. The preprocessor statically computes runtime parameters, such as the message buffer address, for each message-passing operation so that the runtime system can avoid the overhead of computing them.

While a number of scheduling algorithms for distributed memory systems have been studied,[9] they are not directly applicable to the Cell BE processor. This is because of key differences in the architectural characteristics; that is, existing algorithms assume loosely coupled coarse-grain multiprocessors, where each processor has a large local memory but the communication latency between processors is very large. The Cell BE processor, on the other hand, is a tightly coupled fine-grain multicore processor where each SPE has a small local memory but the communication latency between SPEs is very small. These differences have led us to a new clustering approach in our static scheduling algorithm.

The contribution of this paper is twofold. First, we propose a microtask model for the Cell BE processor. It frees programmers from explicit local-store management, which could be a significant burden for them. Second, we propose a novel scheduling algorithm that converts a microtask program into one for a streaming model which the Cell BE processor can execute efficiently.

RELATED WORK

The microtask model is compared with other programming models proposed for the Cell BE processor and similar architectures, and related work in static scheduling algorithms is discussed.

PPE-centric versus SPE-centric programming models

Kahle et al.[2] proposed two approaches to map application programs to the Cell BE processor: function offload and computational acceleration models. The function offload model is a PPE-centric

approach in which the main application program runs on the PPE and performance-critical functions in the program are offloaded to SPEs. Typically, programmers are responsible for identifying functions to be offloaded to SPEs and for modifying the original program in such a way that the main program on the PPE uses SPE functions through a remote procedure call. This model has been popular because programmers can usually port existing single-thread programs without changing their main application logic.

The computational acceleration model, on the other hand, is an SPE-centric approach, where the main program runs on SPEs in parallel, and service functions for SPEs are performed on the PPE. Thus, this model uses SPEs in a more integrated fashion than the function offload model. Typically programmers parallelize the program manually to map it to the PPE and SPEs.

The microtask model can be considered as an SPE-centric computational acceleration model; it allows programmers to parallelize the main program by using MPI. It extends the computational acceleration model through an asymmetric thread runtime model,[2] which allows multiple SPE threads to run on a physical SPE. The preprocessor of the microtask program, furthermore, converts microtask applications to those for a streaming model, which allows the multiple SPEs to execute the computation kernels through which the data flows as a stream. In particular, the static scheduler, a part of the preprocessor, optimizes such a conversion.

Shared-memory versus message-passing programming models

Application programs on the Cell BE processor can use a shared memory or message-passing programming model. When they use a message-passing model, each of the SPEs and the PPE has, from the perspective of the application, a separate memory space. The message-passing model makes sense because DMA operations can efficiently transfer messages between two local stores and also between the system memory and a local store. For example, an SPE can transfer a message directly between user-level buffers on a different local store. Historically, the IBM System/390* coupling facility applied similar techniques to implement message passing.[10]

When applications use a shared-memory model, the PPE and SPEs share the off-chip system memory. SPEs can access the system memory by using DMA operations in a cache-coherent way.[2] The shared-memory approach makes sense because the memory access latency via DMA is comparable with that of L2-cache misses of conventional shared memory multicore processors and also because the PPE and SPEs can share a common effective-address space. To run conventional shared memory programs on SPEs, however, one must modify them because SPEs

■ While conventional microprocessors have a hardware cache to manage a small local store, the Cell BE processor must rely on a software mechanism to manage it. ■

can access the shared memory only by using DMA operations but not by using load/store operations.

Eichenberger et al.[11] implemented an OpenMP compiler and its runtime to provide a shared memory programming model. By utilizing a compiler-controlled data cache, this implementation generates DMA transfers only when a cache miss occurs. It also can be viewed as an extended function offload model, where programmers use OpenMP directives to specify functions to be offloaded from the PPE to SPEs.[2] Those offloaded functions, however, may or may not fit into the local store of the SPE. Thus, the OpenMP implementation employs a compiler-controlled code-partitioning mechanism in addition to the compiler-controlled software data cache; these mechanisms allow the SPE to fetch the overflowed code and data from the system memory. The OpenMP implementation applies compilation techniques to reduce the number of cache directory lookups, which are the major performance overhead caused by the software data cache.[11]

The OpenMP approach is, in fact, quite contrastive with the microtask approach. While the OpenMP approach is based on a shared memory model, the microtask approach is based on a message-passing model. While the OpenMP approach relies on

software-managed data-cache and code-overlay mechanisms to make each task fit into the local store, the microtask approach relies on programmers to partition the computation into a collection of microtasks, each of which fits into the local store. Both approaches rely heavily on compilation techniques, but for different purposes. In the OpenMP approach, compilation techniques are important to reduce the performance impact caused by software-managed data-cache and code-overlay mechanisms, while in the microtask approach, compilation techniques are important to efficiently schedule communications and computations on physical SPEs. Such communications include data transfers caused by context switches.

It is too early to compare the two approaches quantitatively in terms of their programmability and performance for realistic applications. One can perhaps argue that the OpenMP approach attempts to apply a traditional symmetric-multiprocessing programming model to the Cell BE processor by providing compiler-controlled software functions to make up for the lack of certain hardware functions, such as hardware cache memory. The microtask approach also applies the traditional message-passing programming model to the Cell BE processor by providing an efficient message-passing runtime. The execution model for the microtask is, however, more optimized than that for traditional message-passing models because it translates the microtask program into a form that stream processing hardware can execute efficiently.

Stream versus message-passing programming models

Several languages that directly express stream processing have been proposed. The most recent ones include Brook[6] and StreamIt.[7] All these languages define several constructs that allow programmers to explicitly define a set of arithmetic-intensive computation kernels, their communication, and their data parallelism. Although each stream programming language defines a set of different language constructs for a different target hardware structure, they generally share the following two goals.[12] First, they make data and pipeline parallelism visible to the compiler, which can exploit multiple functional units or processing elements. Second, they separate communication from computation to allow the compiler to minimize the performance impact of communication latencies.

To this extent, the microtask model shares these two goals. Owing to the fact that it is based on a message-passing model, application programs can make both data and pipeline parallelism visible to the compiler in the form of tasks and their dependencies through messaging. In other words, explicit communications in the message-passing model make it easier for the compiler to separate communications from computation.

Because message-passing models are more expressive in terms of the application algorithm than stream programming models, they need extra compilation techniques to fully optimize the scheduling of communications and computation. More specifically, message-passing models differ from stream programming models in the following two aspects.

First, the two models are different with respect to the degree that computation is separated from communication. Stream programming models define communication between computation kernels outside the definition of computation kernels and thus separate communication from computation at the language construct level. Message-passing models, on the other hand, generally allow programmers to mix communications and computation in the task definition. Thus, unlike a computation kernel in stream programming models, a task in MPI generally interacts with other tasks to proceed and thus does not represent a unit of computation that can run without interacting with other tasks. The static scheduler for the microtask, consequently, divides each task into a set of basic tasks, which corresponds to the computation kernel in stream programming languages, as mentioned earlier. In other words, stream programming languages require programmers to decompose their application to the computation kernel level, whereas the microtask model requires them to decompose it only to the microtask level, and the static scheduler further decomposes each microtask to the computation kernel level.

Second, the two models are different with respect to exposing data parallelism. In stream programming models, computation kernels and streams naturally represent data parallelism when kernels do not have internal states. In Brook, for example, a kernel call represents a `do-all` parallel loop for each element of input streams.[6] In message-passing

models, on the other hand, an input message generally affects the processing of future input messages through some internal states of the task, and thus input messages cannot be processed independently. Unlike stream programming models, message-passing models typically use a single-program multiple-data (SPMD) programming style[4] to represent data parallelism. In the microtask model, furthermore, the scheduler identifies basic tasks that can be executed independently by examining their dependencies. Internal states in the task actually impose another challenge for the microtask model. Because the number of tasks is typically much larger than that of physical processor cores, a context switch may occur at a basic task boundary. Such a context switch typically requires save and restore operations of internal states as the task context. Thus, one of the important roles that the static scheduler must play is to reduce the number of context switches.

Static scheduling for parallel programs in message-passing models

Kwok et al.[9] compared 27 static scheduling algorithms. Among them, two classes of algorithms—unbounded number of clusters (UNC) and bounded number of processors (BNP)—are most relevant to our discussion. In this section, we discuss potential issues that could occur when we apply those two algorithms to the Cell BE processor.

The UNC method consists of two scheduling phases. In the first phase, the number of processors is assumed to be infinite, and a set of tasks is clustered as a unit of computation on a processor. In the second phase, those task clusters are scheduled onto a finite number of processors. While there are several variations in the clustering algorithm, all of them basically attempt to pack tightly communicating tasks into a single cluster. As a result, each cluster represents a coarse-grain computation assigned for each processor. This approach is suitable for coarse-grain parallel systems, such as clustered workstations. However, it is not suitable for fine-grain multicore processors. This is because coarse-grain tasks can cause frequent context switches, which are relatively expensive for the SPE. Our scheduling scheme is similar to the UNC method in the sense that it includes a clustering phase. Nevertheless, it is significantly different from the UNC method in the characteristics of the clusters; that is, the clustering phase in our approach attempts to identify a set of tasks that can run on multiple SPEs in a gang fashion without causing context switches, where SPEs can communicate with each other efficiently. In contrast, the clustering phase in the UNC method attempts to identify a set of tasks that can run on a single general-purpose processor, where each processor can access a large system memory efficiently. As a result, if one applies the UNC method for our case, clusters would cause frequent context switches, which would consume the limited off-chip bandwidth without exploiting the large on-chip bandwidth among SPEs.

■ From a programmer's perspective, each microtask is simply a small MPI task that fits in the local store. ■

The BNP method, on the other hand, is a list scheduling algorithm. This method computes a priority for each task based on a critical-path length, which can be informally defined as an estimated execution time between the beginning of the task and the end of the last task when the number of processors is infinite. This method selects a task of the highest priority first and statically schedules it at the earliest time that the constraints of the task are met. Many highly parallel applications tend to have multiple critical paths with a similar length. As a result, the BNP method tends to schedule several tasks from different critical paths onto the same physical processor in an interleaved fashion. Thus, it tends to cause frequent context switches between those tasks. This is a serious problem for the SPE because a context switch operation is relatively expensive for SPEs. Our scheduling scheme is advantageous when compared with the BNP method because of our clustering algorithm, which leads to fewer context switches than the BNP method.

MICROTASK PROGRAMMING MODEL

From a programmer's perspective, each microtask is simply a small MPI task that fits in the local store. One special microtask, called the *supportive microtask*, runs on the PPE. The supportive microtask does not have the memory-size restriction for microtasks. While regular microtasks typically perform compute-intensive tasks on an SPE, the

supportive microtask typically performs control-intensive functions to support regular microtasks, such as I/O processing. Theoretically, the microtask could support any MPI APIs as long as the application task and the runtime can fit in the local store. Practically, however, we believe the microtask should limit the support of certain MPI APIs that prohibit or excessively complicate the efficient execution of microtask applications or that are of little use for SPEs. Namely, the current design of the microtask system, at least for now, does not support APIs for one-sided communications (i.e., remote memory accesses), those for parallel I/O operations, and other system calls from microtasks on SPEs. In the current microtask implementation, the supportive microtask on the PPE calls the operating system for I/O operations and other services. We also believe the microtask could take the liberty of extending the API, perhaps as "syntactic sugar" (additions to the API that do not affect its expressiveness but make it more programmer-friendly), if such an extension is extremely useful in terms of the expressiveness or performance of microtask applications.

Decomposing applications to microtasks

The microtask adopts the dynamic process model of MPI-2 as a primary process model;[5] that is, microtask applications start with the invocation of a supportive microtask that runs on the PPE. The supportive microtask can create a set of microtasks by calling `MPI_Comm_spawn()`, which is one of the standard APIs in MPI-2. Each microtask, furthermore, can create another set of microtasks hierarchically by calling `MPI_Comm_spawn()`. Each call to `MPI_Comm_spawn()` creates a set of SPMD microtasks, which typically represent data parallelism. Thus, programmers typically decompose an application into multiple sets of SPMD microtasks and keep decomposing them further, sometimes hierarchically, until each microtask fits into the local store. Such decompositions often cause communications within each set of SPMD microtasks and also with their parent, child, and sibling sets.

MPI defines a concept of communicator that corresponds to a communication context.[5] Each communication API typically takes one communicator as a parameter to identify the communication context. Communications within a group use an *intra*communicator that corresponds to the group, whereas communications between a pair of groups use an *inter*communicator that corresponds to the pair. Each group of dynamically spawned tasks can use `MPI_COMM_WORLD` as a default intracommunicator. MPI-2 provides APIs to construct intercommunicators for two kinds of groups: child/parent and client/server.[5] The latter typically involves a name service, which manages service names (a character string) and associates each service name with an actual task group.

Because communications between sibling groups are very common in microtask applications, as mentioned previously, we have found that the following API makes it simpler to program microtasks with intercommunicators:

```
int uMPI_Connect_task(MPI_Comm comm1,
                      MPI_Comm comm2,
                      MPI_Comm *new_comm)
```

The `uMPI_Connect_task()` API involves three groups of tasks: a parent group and its two child groups. Applications in the microtask model typically use this API in the following scenario. The parent group calls this API by passing two intercommunicators in `comm1` and `comm2`, one for each child group. The two child groups, on the other hand, call this API by passing an intercommunicator with their parent in `comm1`, and a null intercommunicator in `comm2`. When the API call returns, it passes a new intercommunicator between the two child groups in `new_comm`.

In addition to convenience for programmers, this API helps the preprocessor identify the dependency between microtasks and hence helps the preprocessor optimize the scheduling of microtasks. In this sense, this API is similar to language constructs in stream programming languages that define communications between computation kernels. Note that this API does not necessarily impose synchronization overhead among the three task groups at runtime. If the preprocessor can statically identify the dependency between microtasks, it translates MPI APIs to lower-level specialized functions that do not use the intercommunicator. We describe more on how to use this API by showing an example later in this section.

Differences from traditional MPI programming

Although the communication API is basically the same between microtask programs and traditional MPI programs, the programming style is actually

different between the two. Traditional MPI applications consist of a set of SPMD tasks that run in a static process model, where all tasks share the same code and each task is associated with a processor that is physically available when the program starts. This style is convenient for conventional coarse-grain parallel systems. For example, this style makes it possible for programmers to write portable applications easily because they can manage physical resources in the application without depending heavily on the operating system.

Microtask applications, on the other hand, cannot adopt this traditional MPI programming style because each task has a very small memory space. Thus, microtask applications usually consist of multiple sets of SPMD microtasks after they are decomposed on both code and data planes; that is, on a code plane, programmers must divide large applications into multiple sets of application-level functions. Furthermore, on a data plane programmers must divide each application-level function into a set of SPMD microtasks so that the data (as well as the code) of each microtask can fit into the local store.

Microtask program example

Figure 1 shows pseudocode for a microtask program example, a two-dimensional fast Fourier transform (2D FFT), which consists of three sets of microtasks. The two sets, X and Y microtasks, perform a 1D FFT for each dimension, x-axis and y-axis respectively. Each 1D-FFT operation uses a radix-2 Cooley-Turkey method. The third set is the supportive microtask that runs on the PPE to control the rest of the microtasks, feeds the input data to them, and then collects the output data from them. Note that in Figure 1 the supportive microtask executes the `main()` function, while the X and Y microtasks execute `x_main()` and `y_main()`, respectively. This example is intended for illustrative purposes only and is not necessarily an optimized version of a 2D FFT.

This application assumes that the data set for each 1D-FFT operation may not fit into the eight local stores of the chip. Thus, it must be divided into small pieces that can fit into one local store. First, each X microtask is assigned to one of such data pieces to perform several butterfly stages in a Cooley-Turkey method. Each X microtask then exchanges the data with another X microtask to perform one butterfly stage. It continues the data exchange and one-stage butterfly operation until the entire 1D data is transformed. Then it sends the results to a set of Y microtasks, which perform 1D-FFT operations for the y-axis by using the same method. Finally, Y microtasks send out the result to the supportive microtask.

Now we describe the behavior of the microtask runtime for this application. At the beginning of the program, the supportive microtask creates two sets of microtasks by calling `MPI_Comm_Spawn()`. Each call to this API creates a set of tasks and returns an intercommunicator between the supportive microtask and the newly created tasks. The supportive microtask then calls `uMPI_Connect_task()` to create a new intercommunicator between the two sets of microtasks, X and Y. The `uMPI_Connect_task()` API is a collective function, which the three sets of microtasks need to call. After creating the new intercommunicator, the supportive microtask sends the input data to X microtasks and then waits until it receives the result from Y microtasks.

■ The concept of a basic task is analogous to that of the basic block, which consists of straight-line code without any jump or jump targets in the middle. ■

The X and Y microtasks obtain an intercommunicator between the two groups by calling `uMPI_Connect_task()`. Because of this intercommunicator, X microtasks can send intermediate results directly to Y microtasks without asking the supportive microtask for help. In other words, the intercommunicator helps make it easy to allow the preprocessor to analyze the dataflow among microtasks. For example, without using this intercommunicator, X microtasks need to send intermediate results to the supportive microtask, which forwards them to Y microtasks. Thus, the two groups always have to go through the system memory to communicate with each other. When they can directly communicate with each other, however, we can allocate communication buffers on a local store as long as they fit into the local store. In this way, we can reduce the number of DMA transfers between the local store and the system memory to improve system performance.

```
void fft1d(complex *buf_a)
{
  fft1d_local(buf_a);  // local butterfly stages
  for (loop = 0; loop < num_loops; loop++) {
    // exchanging data with a peer task
    MPI_Isend(buf_a, points_per_task, MPI_COMPLEX, peer_task(loop),
                0, MPI_COMM_WORLD, &req[0]);
    MPI_Irecv(buf_b, points_per_task, MPI_COMPLEX, peer_task(loop),
                0, MPI_COMM_WORLD, &req[1]);
    MPI_Waitall(2, req, stat);
    fft1stage(buf_a, buf_b, loop);  // one-stage butterfly
  }
}

int x_main(int argc, char *argv[])  // the main for X microtasks
{
  MPI_Init(&argc, &argv);
  MPI_Comm_get_parent(&parent);
  // creating an inter-communicator between X and Y microtasks
  uMPI_Connect_task(parent, MPI_COMM_NULL, &sibling);
  // receiving the input from the supportive (parent) microtask
  MPI_Recv(data, points_per_task, MPI_COMPLEX, 0, 0, parent, &status);
  fft1d(data);
  for (i = 0; i < points_per_task; i++)  // sending to Y microtasks
    MPI_Isend(&data[i], 1, MPI_COMPLEX, y_dest(i), 0, sibling, &req[i]);
  MPI_Waitall(points_per_task, req, stat);
  MPI_Finalize();
}

int y_main(int argc, char *argv[]) // the main for Y microtasks
{
  MPI_Init(&argc, &argv);
  MPI_Comm_get_parent(&parent);
  // creating an inter-communicator between X and Y microtasks
  uMPI_Connect_task(parent, MPI_COMM_NULL, &sibling);
  for (i = 0; i < points_per_task; i++)  // receiving from X microtasks
    MPI_Irecv(&data[i], 1, MPI_COMPLEX, x_src(i), 0, sibling, &req[i]);
  MPI_Waitall(points_per_task, req, stat);
  fft1d(data);
  // sending the output to the supportive (parent) microtask
  MPI_Send(data, points_per_task, MPI_COMPLEX, 0, 0, parent);
  MPI_Finalize();
}
```

Figure 1
Pseudocode for a microtask program example (2D FFT); Part 1 of 2, regular microtasks

STATIC SCHEDULING ALGORITHM

For those applications for which we can statically construct a task precedence graph in a direct acyclic graph (DAG) form, we can utilize the task precedence information for various optimizations. We call such an application a *static application*. For static applications, a preprocessor can analyze their source code and convert MPI functions in the code to optimized lower-level functions at compile time. The preprocessor has four phases: task graph generation, clustering, scheduling, and runtime parameter generation. We describe each phase in detail.

Task graph generation

In this phase, the preprocessor first divides each microtask into a set of basic tasks, each of which represents a unit of computation that does not cause a context switch in the middle, as described earlier. The concept of a basic task is analogous to that of the basic block, which consists of straight-line code without any jump or jump targets in the middle.

Note that we have no need to restrict the size of the microtask but only that of the basic task but because the basic task is not a visible component for programmers, we impose the size limitation to the microtask. It would be interesting to examine how

```
int main(int argc, char *argv[]) // the main for supportive microtask
{
  MPI_Init(&argc, &argv);
  init_data(data2d);
  // creating X and Y microtasks
  MPI_Comm_spawn("x_main", argv, x_size, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                          &x_comm, errcodes);
  MPI_Comm_spawn("y_main", argv, y_size, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                          &y_comm, errcodes);
  // creating an inter-communicator between X and Y microtasks
  uMPI_Connect_task(x_comm, y_comm, &comm);
  // sending the input to X microtasks
  for (y = 0; y < y_points; y++)
    for (x = 0; x < x_points; x+=points_per_task)
      MPI_Send(&data2d[y*x_points+x], points_per_task, MPI_COMPLEX,
                    dest(x, y), 0, x_comm);
  // receiving the output to Y microtasks
  for (x = 0; x < x_points; x++)
     for (y = 0; y < y_points; y+=points_per_task)
       MPI_Recv(&data2d[x*y_points+y], points_per_task, MPI_COMPLEX,
                    src(x, y), 0, y_comm, &status);
  MPI_Finalize();
}
```

Figure 1
Pseudocode for a microtask program example (2D FFT); Part 2 of 2, supportive microtask

programmability and performance would be affected if we imposed the size limitation on the basic task.

The preprocessor then constructs a task precedence graph, where each node corresponds to a basic task and each edge corresponds to a pair of send and receive operations. ***Figure 2A*** shows an example of a basic task graph. This example illustrates a task graph for a 1D-FFT program. For purposes of explanation, we focus on a part of the 2D-FFT program that we discussed earlier (Figure 1). This 1D-FFT example consists of nine microtasks, eight of them for the main computation and one for supportive operations. In Figure 2A, each square represents a microtask, whereas each numbered circle represents a basic task.

Clustering

After generating a task graph, the preprocessor groups basic tasks into clusters. Each cluster consists of a set of basic tasks that are derived from one or more microtasks, and it can be executed by the Cell BE processor without any context switches. Each cluster must not involve more microtasks than the number of physical SPEs, and it must not have a cyclic dependency on other clusters. Here, we informally denote that Cluster A has a cyclic dependency on other clusters if there is a path that goes out from Cluster A and comes back to Cluster A. The preprocessor first picks up a basic task as a seed of a cluster and then attempts to grow it by adding other basic tasks that have a strong dependency on the cluster. We define the strength of the dependency between a task and a cluster as the amount of communication between the task and the cluster required for exchanging messages and context switches. Each cluster should be able to run in a gang fashion without causing a context switch, and the preprocessor stops the growth of a cluster if any additional basic task causes a context switch in the cluster. Clustering is, in fact, the most important optimization. It greatly affects the overall system performance.

The clustering phase then generates a precedence graph of clusters, which is used in the next scheduling phase. ***Figure 2B*** shows the cluster precedence graph for the basic task graph shown in Figure 2A. In this example, we assume there are three physical SPEs.

Scheduling with a dynamic-programming method

It is known that scheduling problems for series-parallel graphs are solvable in polynomial time for

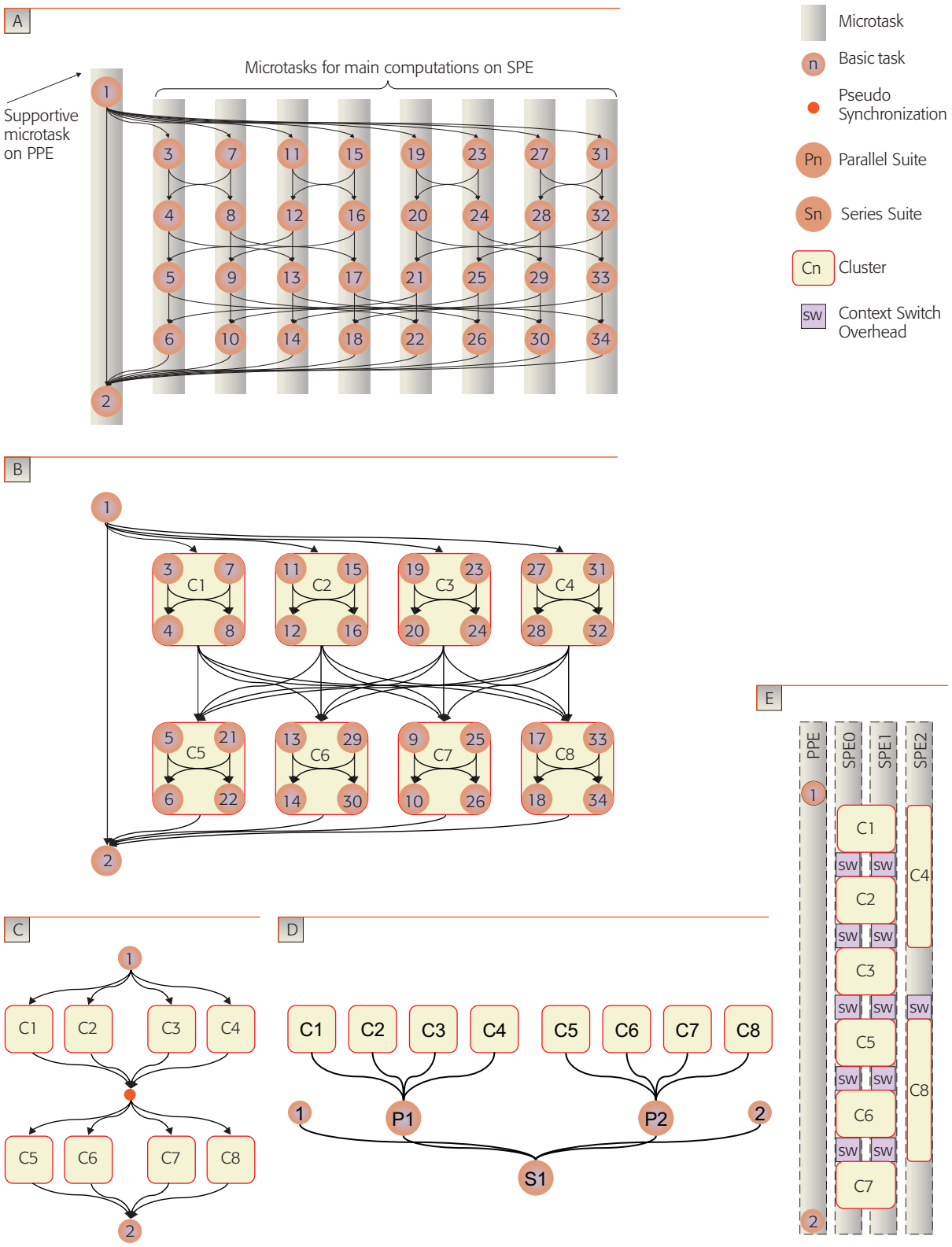

Figure 2
An example scenario of static scheduling (1D FFT); (A) basic task graph; (B) cluster precedence graph; (C) converted series-parallel graph; (D) parse tree; (E) intercluster scheduling result

certain special cases, whereas those for general precedence constraints are nondeterministic polynomial (NP)-hard.[8] Namely, if the task precedence is given by a series-parallel graph and if the longest communication time is much shorter than the shortest computation time, the scheduling problem for series-parallel graphs is solvable in polynomial time for a sufficient number of processors.[8] Our dynamic-programming algorithm solves the scheduling problem by transforming it to a simpler problem. This transformation involves two levels of approximation. First, we approximate the task precedence graph in a series-parallel graph. Second, we bound the search space for scheduling the series-parallel graph by assuming that the scheduling problem for each series and parallel suite is independent. This is discussed in more detail later in this section.

The series-parallel graph has a nested structure of series and parallel graphs, as discussed previously. In a series graph, all nodes need to be scheduled in a series fashion. In a parallel graph, all nodes are independent and can be scheduled in a parallel or series fashion. It is known[13] that any DAGs can be converted into a series-parallel graph by adding some extra edges (dependencies) and by removing transitive edges. When the preprocessor converts the precedence graph of clusters into a series-parallel graph, we limit the number of children to k for each parallel suite so that we can limit the amount of computation in the next step.

In the course of converting the graph, the preprocessor also creates a parse tree of the series-parallel graph. Each leaf node in the parse tree corresponds to a cluster, and each non-leaf node corresponds to a series or parallel suite that consists of its immediate children. ***Figure 2C*** shows the converted series-parallel graph when k is 4 for the cluster precedence graph shown in Figure 2B. ***Figure 2D*** shows the parse tree. Although the preprocessor adds pseudo-synchronization nodes to convert the cluster graph in a series-parallel form (Figure 2C), those pseudo-synchronization nodes are used only for conceptual purposes and do not really cause synchronization operations at runtime.

Once we obtain a scheduling problem in a series-parallel form, it is natural to apply a dynamic-programming method if we can assume each series or parallel suite is an independent subproblem; that is, starting from lowest non-leaf nodes in the parse tree, which have only leaf nodes as their children, we compute an expected execution time when a node is executed by p SPEs for each p ($1 \leq p \leq P$), where P is the number of physical SPEs. For the leaf nodes, we use a brute force method to schedule basic tasks in each cluster. The computation for the intracluster scheduling, however, is bounded because the number of microtasks in each cluster does not exceed the number of physical SPEs, which is eight for the first-generation Cell BE processor. For series suites, we simply add up the minimum execution time of each immediate child when p SPEs execute it. For parallel suites, we compute the execution time for each case when the k immediate children are executed in series, in parallel, or partially in series and in parallel. Then we select the shortest execution time for each p ($1 \leq p \leq P$). The final solution is the minimum execution time of the root node in the parse tree when P SPEs execute the root node. Because we schedule a node (series or parallel suite) by using the intermediate scheduling results only of its immediate children and because we schedule nodes from the lowest level toward the root, we can schedule a node by using the scheduling results for nodes that have been visited. In other words, the preprocessor takes a bottom-up approach for applying a dynamic-programming method by visiting each node only once. Thus, the total computation time is a linear order on the number of clusters.

This linear-order algorithm obviously involves some approximation because the scheduling problem for general series-parallel graphs is known to be NP-hard. In fact, the scheduling problem for each suite is not a totally independent subproblem because of the following two reasons. First, when we schedule a series or parallel suite, the result usually includes some slack time during which one or more SPEs becomes idle. Thus, if we consider each suite as an independent subproblem, we ignore scheduling options to fill such slack time with clusters that are outside the suite. In our current algorithm, we simply ignore the effect of the slack time.

Second, the independence between suites does not hold because the transition time between sequentially scheduled suites may depend on the internal scheduling result of both suites. Suppose two parallel suites, Pa and Pb, are scheduled in back-to-back fashion, and each parallel suite consists of two

gang-scheduled basic tasks; that is, Pa consists of Ta_0 and Ta_1, and Pb consists of Tb_0 and Tb_1. When Pa and Pb are scheduled on two physical SPEs, Ta_0 can be followed by either one of the two basic tasks in Pb (Tb_0 or Tb_1), and Ta_1 is followed by the other basic task in Pb. If Ta_0 and Tb_0 belong to the same SPMD set, the context switch from Ta_0 and Tb_0 needs to swap only the data portion of the context. If not, it needs to swap the entire context on the local store. Thus, the latter case requires a longer context switch time on the SPE than the former case. The transition time from Pa to Pb, therefore, may depend on their internal scheduling results. We, however, simply assume the transition time between any two clusters is constant. Although this assumption is not accurate, it allows us to treat the scheduling problem of each series or parallel suite as an independent problem. In other words, we can use the linear-order scheduling method for intercluster scheduling and can limit the use of an expensive brute force method only for intracluster scheduling.

In our example, shown in Figure 2, the intracluster scheduler obtains the execution time for all clusters when the number of SPEs is one, two, and three. When the intercluster scheduler schedules a parallel suite P1, for example, it chooses the number of SPEs to be allocated to each immediate child (C1–4) by applying the dynamic-programming algorithm. *Figure 2E* shows the result of the intercluster scheduling. In this example, it allocates one SPE for two clusters (C4 and C8), and two SPEs for the rest of the clusters. In this way, this scheduling scheme optimizes the intracluster and intercluster scheduling by applying a dynamic-programming algorithm.

Optimizing runtime message operations

After performing intercluster scheduling, the preprocessor recreates a precedence graph of the basic tasks that have been scheduled on physical processor cores. The task graph represents the execution order of all basic tasks and their dependencies caused by message passing. By using the task graph, the preprocessor assigns a specialized function to each message operation and computes its parameters for optimizing message operations at runtime. To that end, the preprocessor identifies whether it needs a synchronization operation, a context switch, and a message buffer (on a local store or the system memory). We briefly describe each case in more detail.

First, a message operation generally requires a synchronization operation between the sender and the receiver; that is, the receiver needs to confirm a message arrival before fetching the message from a buffer. For certain cases, in fact, the receiver can fetch the message without confirming its arrival because DMA operations with an optional fence semantics can maintain the order of message transfers. The preprocessor identifies such cases in the task graph to reduce the number of runtime synchronization operations.

Second, the preprocessor identifies message operations that cause a context switch by simply scanning the task graph. If two basic tasks, T_a and T_b, are scheduled on the same physical core in a back-to-back fashion, and if T_a and T_b belong to different microtasks, a context switch occurs between T_a and T_b.

Third, the preprocessor determines the location of the message buffer—a local memory or the system memory—for each message operation. The current prototype assigns a buffer on the local memory for intracluster communications; a DMA mechanism transfers a message between SPEs directly. The current prototype simply assigns a system memory buffer for intercluster communications. We plan to improve it by assigning a local-memory buffer for certain intercluster communications.

Finally, the preprocessor can compute all parameters statically for each specialized message function, such as the system memory address of the message buffer (if the communication uses a buffer on system memory), the index of the physical SPE of the peer basic task (if the communication uses a direct SPE–SPE message transfer), and the index of the next microtask to be scheduled (if the communication causes a context switch). Because the preprocessor computes these parameters, the runtime system can transfer messages efficiently, as we describe next.

Runtime organization

This section describes a high-level structure of the runtime system for the current implementation. *Figure 3* shows the memory layout of each SPE. Out of the 256-KB local store, the lowest 16 KB is reserved for the resident runtime, which contains per-SPE data structures, such as synchronization

flags and performance counters. The highest 2KB is reserved to store the register context of the microtask, which consists of 128 128-bit registers. The rest of the 238 KB is left for a microtask and its runtime library. The runtime library contains a message API table and a set of specialized message functions, which we discussed previously. Each table record consists of a pointer to a specialized message function in the runtime library and some parameters for the function. When the microtask calls a message-passing API, such as `MPI_Send()` and `MPI_Recv()`, the runtime library looks up the table and jumps to the specialized function with the parameters supplied in the table record. The runtime library usually increments the index at each table lookup for an API call, so that each message transfer looks up a unique record in the table.

The memory layout of the system memory is similar to that of each SPE except for the following two differences. First, the memory space for the supportive microtask does not have the 256-KB limitation. Second, the system memory contains a large buffer pool of messages and a backing store of all microtask contexts. The preprocessor statically manages the recycling of the buffer pool, which holds the messages that do not fit in the local store.

To illustrate how these runtime components work together, we describe an example in ***Figure 4***. First, in Step 1, a microtask M_a calls `MPI_Recv()`, a blocking receive operation (A), to look up the message API table (B) in the runtime. In this example, M_a needs to switch to another microtask M_b to receive the message. After `MPI_Recv()` calls the specialized function to which the table record points (C), it passes the index of M_b to a resident runtime routine, `syscall()` (D). This routine saves all registers to the register context area in the local store and the local store image to the context backing store of M_a on the system memory (E). In Step 2, it restores the context of M_b from the backing store (F), and starts executing a basic task. When the basic task is completed, in Step 3, M_b calls `syscall()`, which saves the context of M_b (G). In Step 4, the resident runtime then restores the context of M_a (H) and returns to the runtime library. The library then looks up the next table record (I) and jumps to the specialized function to which the table record points (J). In this example, the specialized function directly transfers the message

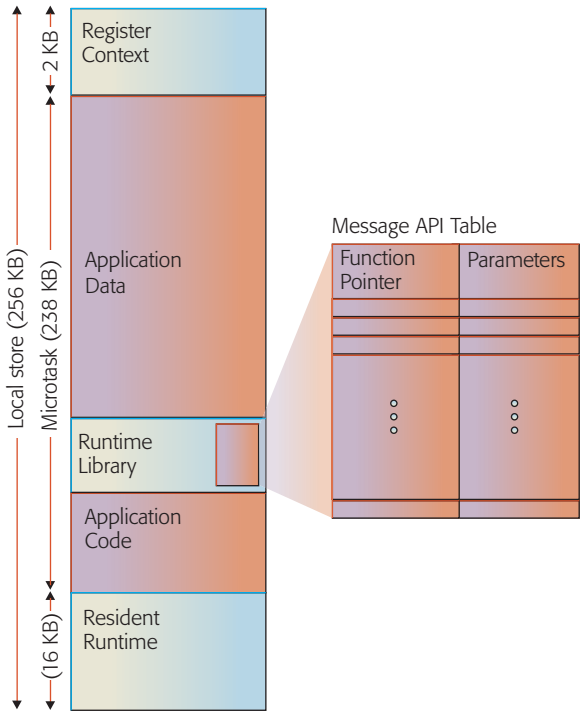

Figure 3
Memory layout of local store for each regular microtask

from a buffer on the system memory without a synchronization operation. It performs a DMA transfer (K) by using the buffer address given by the corresponding table record (I).

Note that each SPE can perform a context switch without communicating with the PPE because the resident runtime can directly swap microtask contexts by accessing their backing store on the system memory. As we discussed previously, we usually schedule a set of microtasks in a gang fashion if they belong to the same cluster. A naive implementation of such gang scheduling requires a barrier synchronization at which each microtask has to wait for other microtasks in the gang set. Such an implementation, furthermore, can cause heavy congestion at the system memory because multiple context switches occur at the same time. Our implementation, on the other hand, does not use a barrier synchronization when a cluster is being scheduled. Instead, it uses a pair-wise synchronization between a sender and a receiver within the same cluster at each message-passing operation. In this way, we can avoid heavy congestion at the system memory by allowing each microtask in a cluster to start execution independently.

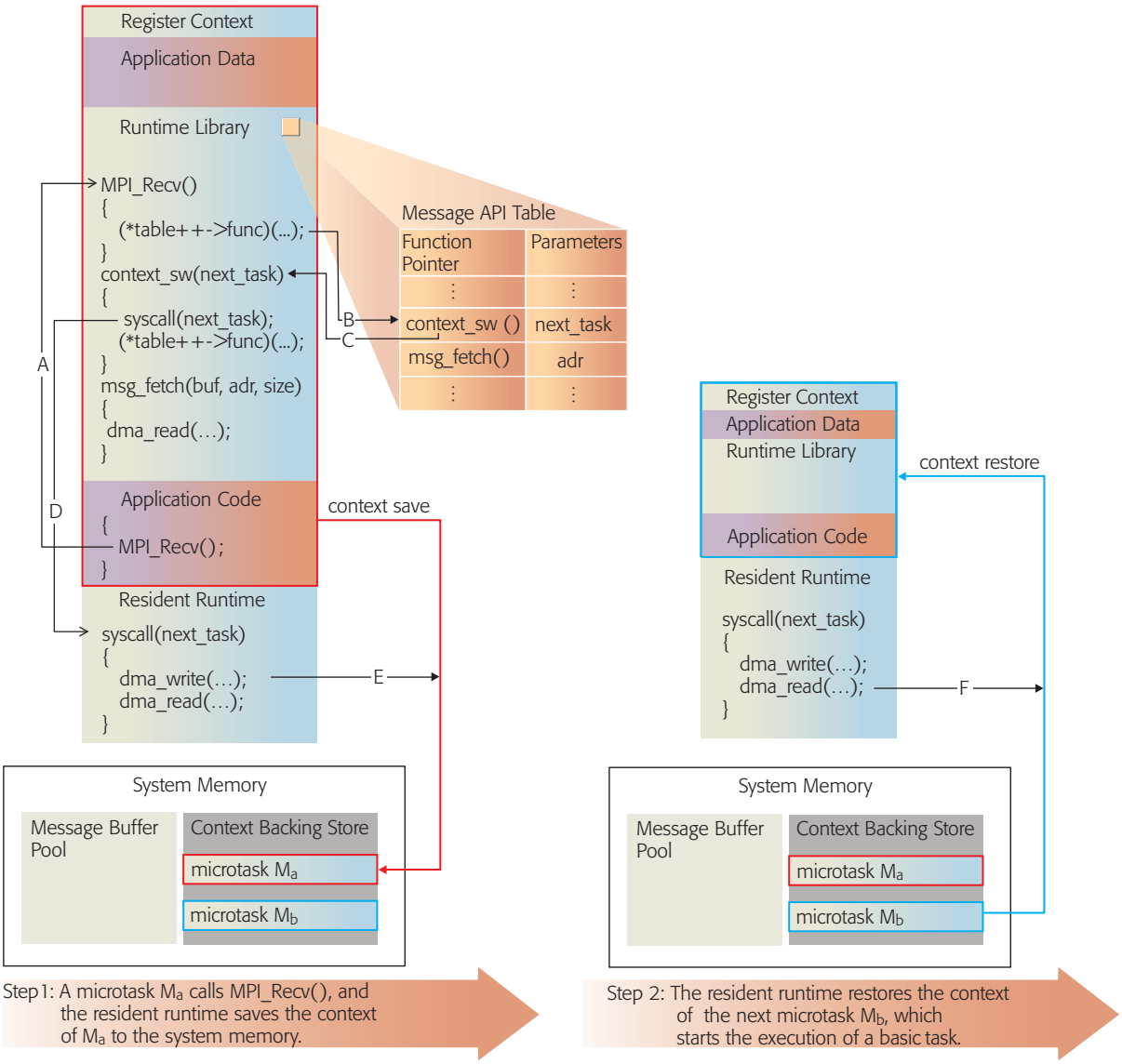

Figure 4
An example scenario of a message-receive operation (Part 1 of 2)

The current implementation with a message API table is reasonably efficient because the table is structured in such a way that the SPE can efficiently access it. Our early experimental results have, in fact, shown that the overhead for table lookups is very small. Nevertheless, we can further reduce the overhead by inlining certain table entries inside the microtask code.

EXPERIMENTAL RESULTS

We have implemented an initial prototype to evaluate the microtask programming model and its scheduling algorithm. For our experiments, we used three well-known computation kernels: an LU decomposition (`LU`), a 1D FFT (`FFT1D`), and a matrix multiplication (`MATMUL`). First, `LU` decomposes a 1024×1024 matrix into an LU form by using 32 microtasks. The program divides the matrix into 32 stripes and assigns each stripe (128 KB) to a microtask. Second, `FFT1D` performs a radix-2 Cooley-Turkey algorithm for 256-KB complex data by using 32 microtasks. The program divides the data into 32 chunks—each 64 KB consisting of 8-KB 8-byte complex numbers—and assigns each chunk to a microtask. Third, `MATMUL` multiplies two 576×576 matrices by using 36 microtasks. The

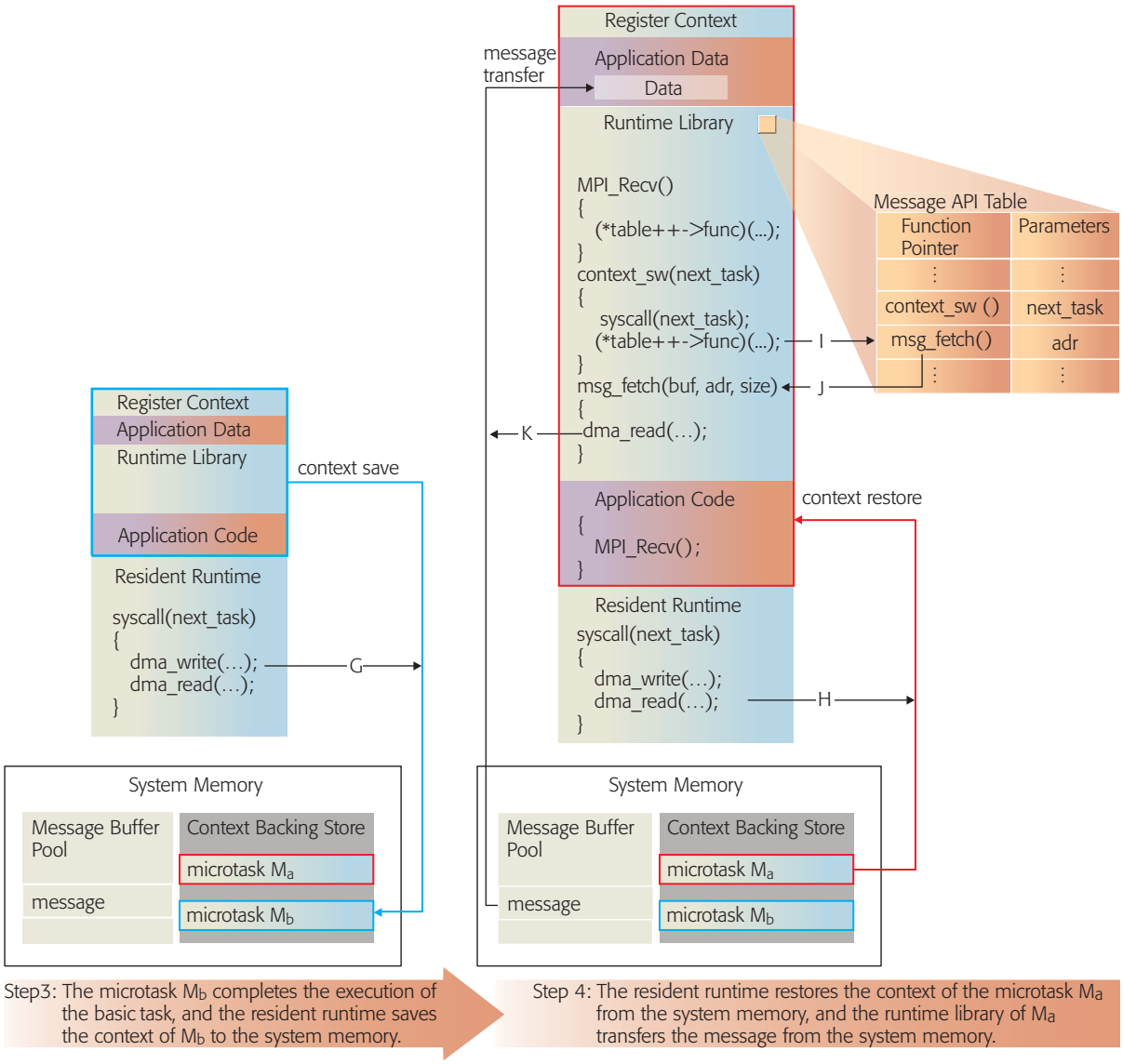

Figure 4
An example scenario of a message-receive operation (Part 2 of 2)

program divides the resultant matrix into 36 96 × 96 submatrices and assigns each submatrix (36 KB) to a microtask. All of these programs use single-precision floating operations to exploit the four-way single-instruction multiple-data (SIMD) engine on each SPE.

Our microtask prototype has certain limitations, the most notable being the current reliance of the preprocessor on programmers to specify some user directives to construct the task precedence graph. A future version of the preprocessor is planned to have a source-code analysis phase that can automatically generate the directives as an intermediate representation of the task precedence graph.

To evaluate the performance advantage of our scheduling algorithm, we have also implemented a well-known critical-path scheduling algorithm[9] for comparisons. As mentioned before, it first computes the critical-path length for each basic task. It maintains a list of runnable basic tasks whose precedence tasks have been scheduled, and it always selects the runnable basic task with the longest critical path in the list. Our algorithm, moreover, uses a simple heuristics to reduce the

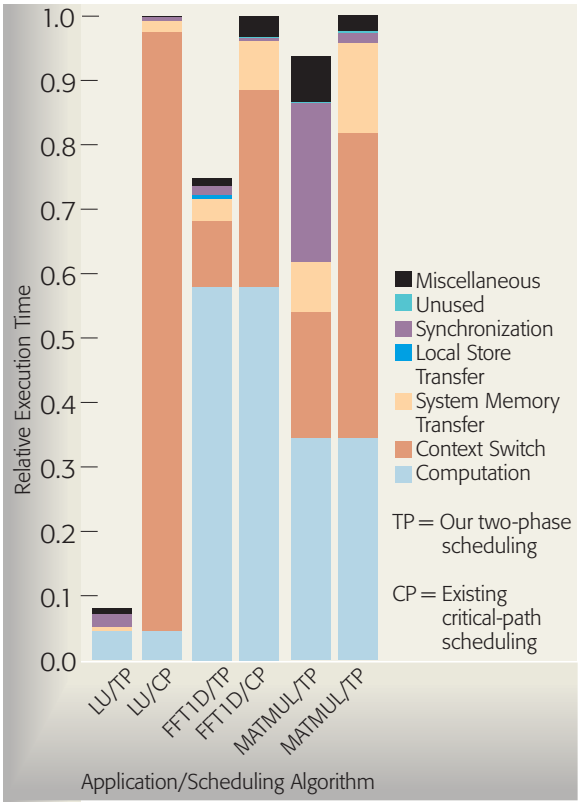

Figure 5
Experimental results

number of context switches; that is, it looks for an SPE whose latest scheduled basic task belongs to the same microtask as the task to be scheduled currently. If there is such an SPE, it schedules the basic task on that SPE to avoid a context switch operation. Otherwise, it schedules the basic task on an SPE that can complete it at the earliest time. In our experiments, this critical-path scheduler uses the same basic task graph for input and the same runtime system as our two-phase (i.e., intracluster and intercluster) scheduler.

We ran the three programs for each scheduling algorithm on a Cell BE processor prototype system. *Figure 5* shows the relative execution time for the three programs. Each execution time is normalized with respect to that for the critical-path scheduling. We break down the relative execution time into seven components: computation, context switch, system-memory transfer, local-store transfer, synchronization, unused, and miscellaneous. Computation indicates the time during which an SPE was busy for actual computations. Context switch indicates the time during which an SPE saved or restored a context. System-memory transfer indicates the time during which an SPE read or wrote the system memory for message transfers via DMA. Local-store transfer indicates the time during which an SPE transferred a message between two local stores. Synchronization indicates the time during which an SPE waited for a message. Unused indicates the time during which an SPE waited for the completion of other SPEs at the end of the program. Miscellaneous indicates the time for any other overhead in the runtime system, such as the message-API table look-ups. The miscellaneous component is negligible (less than 0.2 percent in the relative execution time) for all cases shown in Figure 5. For these experiments, all DMA transfers do not overlap with other computations.

Our experimental results have shown that our clustering algorithm consistently reduces the context switch overhead in the execution time as intended. This reduction is most significant for `LU`. This is because `LU` consists of many basic tasks that can run in parallel. As a result, the critical-path scheduler schedules basic tasks in such a way that the 32 microtasks are interleaved on the eight SPEs in fine granularity. This scheduling pattern results in a large context switch overhead. In our two-phase scheduler, clustering can prevent such a large context switch overhead.

Due to the reduction in the context switch overhead, the total execution time for our two-phase scheduling algorithm is smaller than that for a critical-path scheduling algorithm for all the three programs we examined. The difference in the total execution time between the two algorithms is most significant for `LU` and least significant for `MATMUL`. For `MATMUL`, our clustering algorithm successfully reduces the context switch overhead, but it increases the synchronization overhead. This is because our algorithm creates some clusters with relatively large slack time, which in turn cause load unbalancing among SPEs. This indicates that there are opportunities for improving our clustering algorithm further.

CONCLUSION

Although the Cell BE processor has a very high peak performance, it relies on software control for certain functions that microprocessors typically support in hardware. One such function is local-store management, which could burden programmers with significant effort if they are entirely responsible. The

microtask model we proposed here provides a unique programming model for the Cell BE processor to free programmers from local-store management by allowing the preprocessor and the runtime system to optimize the scheduling of communications and computation.

In addition to providing a well-known message-passing programming model, the microtask model enables the preprocessor to convert the program into one for a streaming model, in which both task contexts and messages stream through processor cores. The preprocessor performs this conversion by dividing each microtask into several basic tasks, each of which represents a unit of computation that does not communicate with other basic tasks in the middle. Each basic task also corresponds to a computation kernel in stream programming models. Finally, the preprocessor schedules the computation and communications of basic tasks in such a way that the Cell BE processor can execute them efficiently. For that purpose, we propose a novel scheduling algorithm suitable for the unique architecture of the Cell BE processor.

We have implemented an initial prototype of the microtask preprocessor and the runtime system. Our early experiments have shown some encouraging results; our scheduling algorithm consistently reduced the context switch overhead for the programs that we examined. The results also indicated opportunities to enhance our clustering algorithm for further performance improvements. Thus, the microtask model should make it easier for programmers to exploit the high peak performance of the Cell BE processor.

CITED REFERENCES

1. D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor," *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'05)*, San Francisco, CA (2005), pp. 184–185.
2. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research & Development* **49**, No. 4/5, 589–604 (2005).
3. H. P. Hofstee, "Power Efficient Processor Architecture and The Cell Processor," *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, San Francisco, CA (2005), pp. 258–262.
4. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd Edition), MIT Press, Cambridge, MA (1999).
5. W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge, MA (1999).
6. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics* **23**, No. 3, 777–786 (2004).
7. W. Thies, M. Karczmarek, M. Gordon, D. Z. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe, *StreamIt: A Compiler for Streaming Applications*, Technical Memo LCS-TM-622, MIT Laboratory for Computer Science, Cambridge, MA 02139 (2001).
8. P. Chrétienne and C. Picouleau, "Scheduling with Communication Delays: A Survey," in *Scheduling Theory and its Applications*, P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, Editors, John Wiley & Sons, Chichester, United Kingdom (1995), pp. 65–90.
9. Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys* **31**, No. 4, 406–471 (1999).
10. J. M. Nick, B. B. Moore, J. Y. Chung, and N. S. Bowen, "S/390 Cluster Technology: Parallel Sysplex," *IBM Systems Journal* **36**, No. 2, 172–201 (1997).
11. A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing Compiler for the CELL Processor," *Proceedings of the 14th International Conference on Parallel Architecture and Compilation Techniques (PACT'05)*, Saint Louis, MO (2005), pp. 161–172.
12. F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz, "The Stream Virtual Machine," *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, Antibes Juan-les-Pins, France (2004), pp. 267–277.
13. A. González-Escribano, A. J. C. van Gemund, and V. Cardeñoso-Payo, "Mapping Unstructured Applications into Nested Parallelism," *Lecture Notes in Computer Science* **2565**, pp. 407–420 (2003).

Accepted for publication June 30, 2005.
Published online January 11, 2006.

Moriyoshi Ohara
IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamatoshi, Kanagawa-ken, 242-8502, Japan (ohara@jp.ibm.com). Dr. Ohara received a B.S. degree in mathematical engineering from the University of Tokyo in 1986, and M.S. and Ph.D. degrees in electrical engineering from Stanford University in 1992 and 1996, respectively. He joined the IBM Tokyo Research Laboratory in 1986 and has

worked on multiprocessor systems, ultra high-resolution display systems, and performance simulation tools. Dr. Ohara's current research interests include high-performance parallel architectures and microprocessor architectures.

Hiroshi Inoue
IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamatoshi, Kanagawa-ken, 242-8502, Japan (inouehrs@jp.ibm.com). Mr. Inoue received B.S. and M.S. degrees in system design engineering from Keio University in 2000 and 2002, respectively. He joined the IBM Tokyo Research Laboratory in 2002 and has worked in the area of optimization techniques for the Cell Broadband Engine™ architecture and the PowerPC® architecture. His research interests include high-performance architectures and optimization techniques for high-performance computing workloads.

Yukihiko Sohda
IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamatoshi, Kanagawa-ken, 242-8502, Japan (sohda@jp.ibm.com). Dr. Sohda received B.S., M.S., and Ph.D. degrees from the Tokyo Institute of Technology in 1998, 2000, and 2003, respectively. He joined the IBM Tokyo Research Laboratory, where he has worked on Web-service caching. His research interests include high-performance parallel architectures.

Hideaki Komatsu
IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamatoshi, Kanagawa-ken, 242-8502, Japan (komatsu@jp.ibm.com). Dr. Komatsu received B.S. and M.S. degrees in electrical engineering from Waseda University in 1983 and 1985, respectively, and a Ph.D. degree in computer science in 1998. He joined the IBM Tokyo Research Laboratory in 1983, where he has carried out research activity in many areas. His research interests include compiler optimization techniques and high-performance architectures.

Toshio Nakatani
IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamatoshi, Kanagawa-ken 242-8502, Japan (nakatani@jp.ibm.com). Dr. Nakatani received a B.S. degree in mathematics from Waseda University in 1975 and M.S.E. and M.A., degrees from Princeton University in 1985. He received a Ph.D. degree in computer science from Princeton University in 1987, when he also joined the IBM Tokyo Research Laboratory. Dr. Nakatani is currently an IBM Distinguished Engineer and manager of the Systems Group. His research interests include computer architectures, optimizing compilers, and algorithms for parallel computer systems. ■