Scalable Inter-Vehicular Applications

Jonathan J. Davies and Alastair R. Beresford

Computer Laboratory, University of Cambridge,
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
{jjd27,arb33}@cam.ac.uk

Abstract. Many pervasive inter-vehicular applications involve the collation, processing and summarisation of sensor data originating from vehicles. When and where such processing takes place is an explicit design-stage decision. Often some processing occurs on vehicles, and some on backend servers, but it is hard for the programmer to optimise this distribution for feasibility or performance. This paper investigates automated task assignment: we define a computational model which captures data aggregation and summarisation explicitly, allowing a compiler to automatically optimise the assignment of processing tasks to particular vehicles and servers. Our model allows a compiler to apply program transformations to data processing, which can further improve task assignment.

Modern motor vehicles contain a plethora of on-board computing equipment. Today's cars have a variety of microprocessors governing diverse aspects of the vehicle's operation. We believe that trends in decreasing power requirements, size, and cost of manufacture mean that in future we can expect vehicles to provide embedded computing platforms supporting the execution of general applications. As cars become increasingly connected—to each other and to the Internet—these applications will evolve beyond disconnected intra-vehicle applications and will help to improve the safety, efficiency and comfort of using transport [1]. This vision of communicating vehicles will enable applications involving multiple participants, such as:

Collection of vehicle position data. Known as *floating car data*, information regarding the locations and velocities of vehicles using the road network can be used to identify levels of road congestion and used as input to journey-time prediction applications [2].

Real-time weather map. Most modern vehicles already contain thermometers. If data from vehicles' onboard weather sensors could be aggregated, a real-time weather map of high resolution could be composed.

Real-time road map updates. Traditional techniques for updating road maps involve manual surveying and data entry. Timely integration of changes to the road network can instead be done automatically based on vehicles' location traces [3].

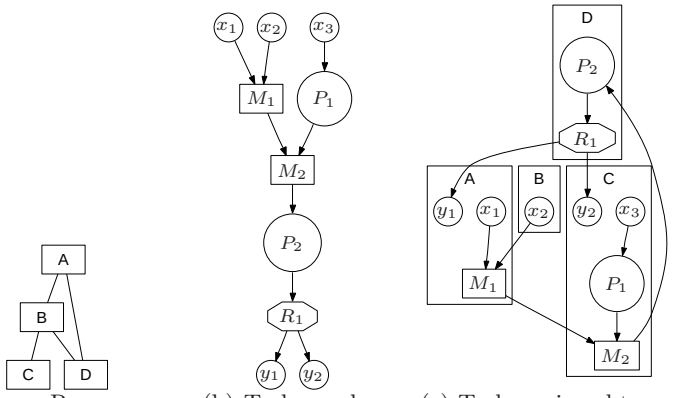
Road hazard detection. Acceleration data collected from vehicles containing accelerometers can be used to build a map of road hazards by noting points in the road network where many vehicles have been found to swerve or brake sharply [4].

Such inter-vehicular applications collate, process and summarise raw sensor data from a large number of vehicles, and typically the greater the number of vehicles involved, the more useful the application becomes. A given application might be written to execute in one of a variety of different architectural configurations. At one end of the spectrum is the fully centralised approach, where all of the source data is transferred to a single processing node. At the other end is the fully decentralised, or peer-to-peer, approach, where there is no infrastructural support. Vehicular applications are not readily-suited to a centralised model of computation because this approach does not scale well and requires ubiquitous network coverage. Fortunately, most applications are inherently parallelisable by partitioning their input data into subsets which can be processed independently. For example, the data set could be decomposed into geographical subsets, with each containing data concerning a particular spatial region.

Today, a programmer writing an inter-vehicular application must manually define where the processing of sensor data takes place. This paper describes an alternative strategy, allowing programmers to write applications which can be distributed *automatically*, therefore potentially at run-time and with a greater degree of optimality because they do not need to rely on design-stage assumptions. More specifically, we devise a computational model which captures the notion of data aggregation or summarisation in an application. This permits the automation of *program transformations*, where the order in which data processing and distribution takes place in an application can be rearranged by a compiler, thereby allowing the exploration of many different software architectures. In addition, we take inspiration from work in parallel and distributed computing and explore how, for a given application with a specific software architecture, the application might be automatically distributed and executed across a heterogeneous network of on-board computers in vehicles and backend servers. We also explore what optimality might mean in the context of vehicular networks, and describe a few metrics on which we can measure and optimise the configuration of an application. Finally, we describe our experience implementing a tool which applies program transformations to an application described in our computational model and automatically simulates the distribution of such an application across many computing resources.

1 Computational Model

It is necessary to be able to define an application and the topology of the network in which it is to be executed at a given instant in time. We model the former (the software) with a *task graph* and the latter (the hardware) with a *computation resource graph*. The task graph is weighted with values indicating the application's requirements, whilst the computation resource graph is weighted with values indicating the hardware's capabilities. We will describe how each is modelled, in turn. We will then turn to the question of how to determine the best computation resource on which to execute each task.



(a) Resource topology graph (b) Task graph (c) Tasks assigned to resources

Fig. 1. Example graphs

1.1 Modelling the Network Topology

At any point in time, the topology of the network of available computation resources is static. It can be modelled by a weighted graph $G_p = (E_p, V_p)$. The set of vertices, V_p, model the processing nodes; the edges, E_p, model direct communication links between nodes. An example is shown in Fig. 1(a). The processing nodes, which have local memory, are not assumed to be homogeneous in their processing power.

If we assume that all the communication links are symmetric, the graph can be undirected. The transitive closure of the graph indicates which nodes can directly or indirectly communicate with any other nodes.

The resource graph's vertices are weighted with values describing their computational characteristics, such as processor speed. The edges are weighted with values characterising the link, such as maximum throughput or latency.

1.2 Modelling the Algorithm

An application's algorithm can be described by a directed weighted graph $G_t = (E_t, V_t)$ called the *task graph*. The set of vertices, V_t, are the tasks and the edges, E_t, indicate the direction of data flow between tasks. A task is a set of instructions which must be executed sequentially on a single processor. An edge (v_1, v_2) indicates that task v_2 receives the output of task v_1, and that v_2 cannot commence execution until the execution of v_1 is complete. An example is shown in Fig. 1(b).

We express algorithms in terms of five types of task nodes:

Source nodes are points where data is produced. These can be thought of as functions of type `unit` $\rightarrow \alpha$, for some type α.

Sink nodes are points where data is consumed. These can be thought of as functions of type $\alpha \rightarrow \texttt{unit}$, for some type α.

Processing tasks are functions which transform a tuple of inputs into a tuple of outputs. In general, they have type $\alpha \times \beta \times \ldots \rightarrow \gamma \times \delta \times \ldots$.

Merge tasks are a special type of processing task which are commutative, associative, binary functions with type $\alpha \times \alpha \rightarrow \alpha$, for some α.

Replication tasks are a special type of processing task which have type $\alpha \rightarrow \alpha \times \alpha$ and additionally where the two outputs are each identical to the input. Hence, it is not possible to modify the data in a replication task.

Identifying merge tasks as a special class of processing tasks is particularly important in applications involving large numbers of inputs, such as those which are the focus of this paper. Because of the wealth of input data, it is usually necessary to be able to aggregate data into a significantly smaller amount of information to make their processing computationally feasible.

Some examples of simple merge tasks are set union, addition and maximisation. In order to express an arithmetic mean of multiple values in terms of a binary function, we must keep track of both the numerator and denominator in the calculation, otherwise we will lose track of the number of items which have contributed to the mean. The merge task is therefore an operation on a pair of values of type $\texttt{real} \times \texttt{int}$ and can be expressed as $M((a_n, a_d), (b_n, b_d)) = (a_n + b_n, a_d + b_d)$. The value of the mean is yielded by a subsequent processing task with type $\texttt{real} \times \texttt{int} \rightarrow \texttt{real}$, $P(a_n, a_d) = \frac{a_n}{a_d}$.

The vertices of the task graph are weighted with values relating to the computational requirements of the application. The graph's edges are weighted with values relating to the characteristics of the data flow between tasks.

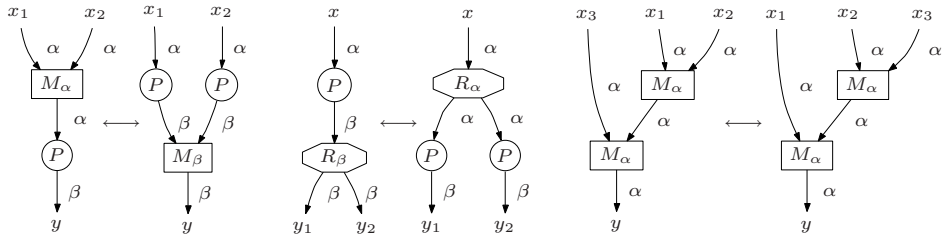
2 Program Transformations

For some programs, it is possible to express the graph of tasks in a variety of semantically-equivalent ways. For example, an algorithm to compute the exponential of the sum of three numbers could be equivalently expressed as the product of exponentials: $e^{x+y+z} = e^x e^y e^z$. The formula e^{x+y+z} is an additive *merge* of the three numbers x, y and z followed by an exponentiation *processing* task; a transformation of the formula gives $e^x e^y e^z$ which is three exponentiation *processing* tasks followed by a multiplicative *merge* of the resulting numbers.

We have defined four transformations which can be applied on any sub-graph of a task graph. After the application of one or more of these transformations, the task graph may be better suited to efficient execution by processors in a particular network. Section 3 will describe the ways in which this can be measured.

2.1 Exchange of Merge and Processing Tasks Transformation

The exponentiation example is an instance of a program transformation involving the exchange of merge and processing tasks which preserves semantic



(a) Exchange of merge and processing tasks

(b) Exchange of processing and replication tasks

(c) Associativity of merge tasks

Fig. 2. Three program transformations (task graph edges labelled with types)

equivalence. Rather than merging some inputs and processing the result, we can process each input individually and merge the results. Figure 2(a) depicts this transformation for a unary processing function of type $\alpha \to \beta$.

Firstly, it is notable that there are more tasks on the right side of the transformation than on the left. This means that the total amount of work required may be different after the transformation is applied, depending on the sizes of tasks M_α and M_β. Moreover, there is one processing task $P : \alpha \to \beta$ on the left and there are two such tasks on the right. Furthermore, the merge task required on the left deals with values of type α and the merge task required on the right deals with values of type β. In the exponentiation example, these functions were addition and multiplication.

Depending on the relative sizes of values of types α and β, the volume of data flow may be affected by this transformation. If values of type β are significantly smaller than values of type α then early processing to β is most favourable, so the volume of data flow is smaller on the right.

2.2 Exchange of Processing and Replication Tasks Transformation

Similarly, rather than performing some processing and then replicating the result, we can replicate the input and process each replica individually. This transformation is depicted in Fig. 2(b).

As above, there is a difference in the number of processing tasks before and after the transformation. On the right, there are two such tasks. As before, depending on the relative sizes of values of types α and β, the volume of data flow may be affected by the transform. If values of type α are significantly smaller than values of type β then late processing to β is most favourable, so the volume of data flow is smaller on the right.

2.3 Merge and Replication Transformations

Two transformations follow directly from the associativity of merge functions and the equivalence of outputs from replication tasks. These transformations

are useful to alter how the merging or replication of a large number of values takes place in a distributed manner. Figure 2(c) depicts the transformation for merge tasks; the transformation for replication tasks is analogous.

3 Execution Strategy

Once we have a model of the application in terms of its constituent tasks and flow of data, and a model of the topology of the network of processors that could execute the application, it remains to define where each task is to be executed.

An assignment function $A : V_t \to V_p$ maps tasks to processing nodes, indicating where in the network each task should be executed. Source and sink vertices in the task graph must be mapped to the particular nodes in the network where data is produced and consumed, respectively. Other tasks can be mapped to network nodes which are reachable from source nodes and from which sink nodes are reachable via communication links. An example of an assignment function is shown pictorially in Fig. 1(c), where M_1 is mapped to processor A, P_1 and M_2 are mapped to processor C, and P_2 and R_1 are mapped to processor D.

The decision about which nodes to use affects the efficacy of the assignment. It impacts on the duration of execution of the algorithm; the privacy of the originators of the data; the amount of network bandwidth consumed; and a variety of other factors. The efficacy of the assignment can be described quantitatively by a *cost function* specific to each application. A cost function $C : G_t \times G_p \times (V_t \to V_p) \to \mathbb{R}$ is a function of an assignment function yielding a real number indicating the cost of the assignment. Applications will use a cost function which embodies the trade-offs they desire between relevant metrics. For example, one application may express in its cost function the policy that only a total execution time of less than two minutes is acceptable, and that minimising the use of network bandwidth is the next most important concern. Another application may seek to minimise total execution time at the expense of all other metrics.

Several candidate metrics for evaluating an assignment function are relevant to many applications:

Total execution time. The duration of time elapsed from the start of the algorithm's execution to the result being delivered to the final recipient.

Quality of result. Thus far, we have assumed that any transformations applied will maintain semantic equivalence of the algorithm. In the exponentiation example, we required the merge functions—addition and multiplication—to be appropriate to maintain this equivalence. But, in the case that a merge function for one data type is not an exact analogue of the one used for the other, the exchange of merge and processing tasks transformation (Sect. 2.1) no longer maintains semantic equivalence, and the algorithm's output approximates the true result. For example, a merge task which sums values followed by a processing task which quantises the sum is approximated by summing quantised values. The value of this metric will relate to the accuracy of the approximation.

Privacy. The level to which the privacy of the originators of the input data is respected is important in applications where personally-identifiable data is processed. The value of this metric could relate to an observer's view of the number of individuals who could have a particular identity.

Energy consumption. Another useful metric is the energy consumption caused by the execution of particular tasks. By associating with each processor a value indicating its power consumption, the total energy consumption for a given assignment function can be calculated.

4 Automatic Task Assignment

In a system where the available resources change dynamically, such as when vehicular resources are utilised, the particular resources which are available for an application to use are not known until run-time. Thus, the assignment of tasks to processors cannot take place at the same time that the application is defined. *Automatic* task assignment is thus required, and this implies a need for a programming language and compiler which can perform this when the program is about to be run.

The compiler must be able to split the program into constituent tasks; determine the optimal mapping with respect to the program's cost function (which may involve the use of some program transformations); distribute the tasks to their processors, ensuring that they can communicate with each other and are able to deal with failures.

4.1 Implementation

We have developed a prototype framework to investigate the feasibility of automatic task assignment. As input, it accepts a description of the computation resource graph and an application's task graph in text format, along with a cost function. The weights of the graphs' nodes and edges that are used by the cost function are also specified.

The framework can automatically derive an optimal assignment function using an exhaustive search, or find a near-optimal assignment function in polynomial time using an approach which involves choosing a reasonable initial assignment and then improving it incrementally by changing the assignment of the tasks until no improvement can be found. The transformations described in Sect. 2 are also applied automatically and the cost of assigning the resulting task graph is compared with that of original graph. The framework also allows a user to assign tasks to processors manually through a graphical front-end, depicted in Fig. 3, which uses *Graphviz* graph layout software to visualise the graphs. It does not execute the application, but merely simulates the effect of the chosen assignment function.

The implementation has highlighted the necessity of adopting a sub-optimal search technique: the sheer size of the search space renders the search for an optimal assignment infeasible for even very small task and resource graphs. However,

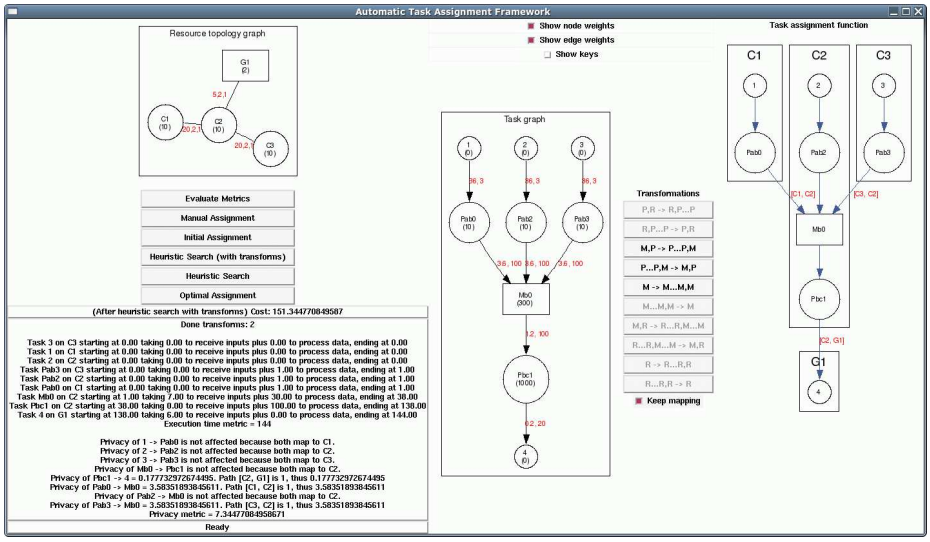


Fig. 3. Prototype framework interface

it has also become clear that implementing a suitable heuristic on which to base the search is challenging. It is difficult to model the effect on the cost of either applying a transformation or modifying the assignment since there are so many variables; this means that traditional techniques for solving the global optimisation problem, such as branch-and-bound and simulated annealing, are hard to apply.

5 Related Work

In recent years, with the emergence of grid computing, the advent of network processors, and the amount of processing possible in wireless sensor networks, there has been a considerable rise in the level of interest in task assignment. In its usual form, grid computing differs from computation in vehicular networks in several characteristics. For example, it is usually the case that the processors comprising a grid, or the data that is processed, are owned by a single entity, so privacy is of no concern. Furthermore, grids may have different priorities regarding the desire to balance load evenly across available processors. Multi-core network processors can be thought of as distributed systems on a single chip, consisting of a matrix of independent processors, each with local memory, sharing a communication bus. The challenges faced in designing compilers for these systems are similar to those examined in this paper. The problems associated with using a global, shared address space have led to suggestions such as the use of linear types [5]. Ennals et al. have devised a set of program transformations which exploit linear types as the programmer expresses the task assignment function [6]. Major differences between network processors and the large-scale distributed systems

considered in this paper are that the processors are arranged systematically, are powered and controlled by a single entity, and have predictable communication links and network behaviour.

Various research has been undertaken into automatically off-loading computation from resource-constrained devices with a view to minimising energy consumption. Several frameworks for off-loading the processing of expensive functions have been devised [7, 8]. Ou et al. have implemented an automatic task partitioning at Java bytecode level to achieve similar goals [9]. Kumar et al. present work in (non-automatic) task assignment [10] in a sensor network environment containing two classes of processor. J-Orchestra [11] is an automatic partitioning system used for splitting up ubiquitous computing applications; the Titan framework [12] has been developed to aid in dynamically reconfiguring which task is allocated to which sensor node.

Traditional approaches to task assignment make use of directed task graphs. Kwok and Ahmad's extensive survey of static task assignment algorithms [13] describes 27 algorithms for scheduling directed task graphs on homogeneous multi-processor systems, but the authors highlight that little work has been done in task assignment for heterogeneous systems, the subject of this paper. Casavant and Kuhl have produced a taxonomy and classify various algorithms against it [14]. Algorithms for task assignment are NP-complete in all but a few restricted cases [15] meaning that it is infeasible to computationally determine the optimal assignment.

The use of program transformations for optimising performance is well-established. For example, compilers typically optimise for execution time or memory footprint by performing semantics-preserving transformations. In databases, there are often many different ways of processing data to formulate the result of a query; it is the job of a query optimiser to choose the optimal approach, which involves rewriting the query into a more efficient form [16]. However, to the best of our knowledge, there is no prior work in compiler theory which makes use of transformations which do not preserve semantics.

6 Conclusions

Traditionally, when designing an application to collate, process and summarise sensor data from a large number of vehicles, a programmer must manually define where these tasks will be executed. We have proposed a strategy whereby the assignment of program tasks to processors is done automatically. The expression of a program in terms of merge, process and replication tasks mean that certain program transformations can be applied automatically by a compiler to allow a more optimal assignment.

Whilst this work was motivated by the problems faced in the implementation of applications involving vehicles, it has a broader applicability to other ubiquitous computing scenarios, in particular to wireless sensor networks.

Further work will examine how the movement of vehicles can be represented in the model, perhaps by defining several vehicles in a spatial area as a single

processing unit. We will also consider whether the optimisation of task assignment can be effected on a local scale rather than globally. We plan to continue to explore the ideas described in this paper by implementing a task partitioning and assignment engine to distribute and execute applications described in an augmented version of the Java programming language.

7 Acknowledgments

The authors thank Samuel Kounev, Andrew Rice, David Cottingham, Tom Craig and Ripduman Sohan for their helpful comments and suggestions; and in particular Andy Hopper for his financial support and Alan Mycroft for the exponentiation example.

References

1. Cottingham, D.N., Davies, J.J.: A vision for wireless access on the road network. In: Proc. WIT 2007, Technische Universität Hamburg-Harburg (2007) 25–30
2. Day, P., Wu, J., Poulton, N.: Beyond real time. ITS International **12**(6) (2006) 55–56
3. Davies, J.J., Beresford, A.R., Hopper, A.: Scalable, distributed, real-time map generation. IEEE Pervasive Computing **5**(4) (2006) 47–54
4. Gruteser, M., Grunwald, D.: Anonymous usage of location-based services through spatial and temporal cloaking. In: Proc. MobiSys 2003, ACM Press (2003) 31–42
5. Ennals, R., Sharp, R., Mycroft, A.: Linear types for packet processing. In: Proc. ESOP 2004. Volume 2986 of LNCS. (2004) 204–218
6. Ennals, R., Sharp, R., Mycroft, A.: Task partitioning for multi-core network processors. In: Proc. CC '05. Volume 3443 of LNCS. (2005) 76–90
7. Kremer, U., Hicks, J., Rehg, J.H.: A compilation framework for power and energy management on mobile computers. Technical Report DCS-TR-446, Rutgers University (2001)
8. Li, Z., Wang, C., Xu, R.: Computation offloading to save energy on handheld devices: A partition scheme. In: Proc. CASES '01, ACM Press (2001) 238–246
9. Ou, S., Yang, K., Liotta, A.: An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In: Proc. PERCOM 2006. (2006) 116–125
10. Kumar, R., Tsiatsis, V., Srivastava, M.B.: Computation hierarchy for in-network processing. In: Proc. WSNA '03, ACM Press (2003) 68–77
11. Liogkas, N., MacIntyre, B., Mynatt, E.D., Smaragdakis, Y., Tilevich, E., Voida, S.: Automatic partitioning for prototyping ubiquitous computing applications. IEEE Pervasive Computing **3**(3) (2004) 40–47
12. Lombriser, C., Roggen, D., Stäger, M., Tröster, G.: Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. In: Kommunikation in Verteilten Systemen (KiVS), Springer Berlin Heidelberg (2007) 127–138
13. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM CSUR **31**(4) (1999) 406–471
14. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. on Soft. Eng. **14**(2) (1988) 141–154
15. Fernández-Baca, D.: Allocating modules to processors in a distributed system. IEEE Transactions on Software Engineering **15**(11) (1989) 1427–1436
16. Ioannidis, Y.E.: Query optimization. ACM CSUR **28**(1) (1996) 121–123